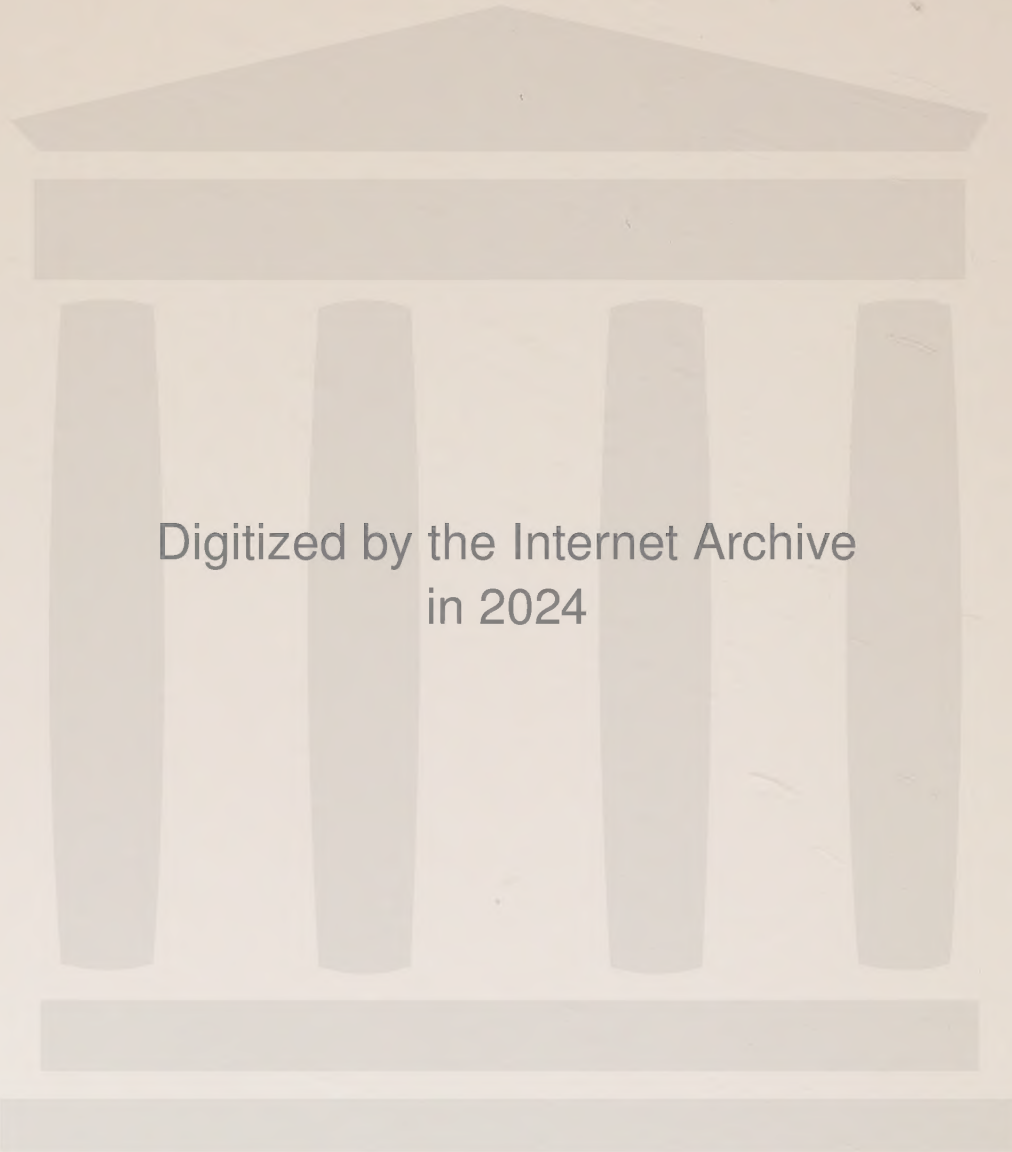


BUILDING APPLICATIONS USING A 4GL

**With
Examples
from
INFORMIX-
4GL**

Mark G. Sobell



Digitized by the Internet Archive
in 2024

<https://archive.org/details/buildingapplicat0000sobe>

BUILDING APPLICATIONS USING A 4GL

**With
Examples
from
INFORMIX-
4GL**

Other books by Mark G. Sobell

A Practical Guide to the UNIX System
(Benjamin/Cummings 1984)

A Practical Guide to UNIX System V
(Benjamin/Cummings 1985)

BUILDING APPLICATIONS USING A 4GL

Second Edition

With
Examples
from
**INFORMIX-
4GL**

Mark G. Sobell

Copy Editing: Rhoda Simmons
Cover Design: Clennon Associates
Illustration: Al McCahon
Production: Sobell Associates
Typesetting: Sobell Associates
Word Processing: Keywords Composition

This book was typeset using the tools of the UNIX system. Drafts were printed on a Xerox 2700 laser printer using **xroff**, which is device-independent **troff** plus a proprietary post-processor from Image Network, Inc. Production simulation of typeset output was generated by Image Network's simulation program and output on a 2700 laser printer. The book was set using Century Schoolbook and Geneva font families on an Autologic APS-5 phototypesetter using device-independent **troff** and Image Network's device driver.

Lotus 1-2-3 is a trademark of Lotus Development Corporation.
IBM is a trademark of International Business Machines.
INFORMIX is a registered trademark of Informix Software, Inc.

Copyright © 1986 by Mark G. Sobell

All rights reserved. No part of this work covered by the copyright hereon may be reproduced, transmitted, or used in any form or by any means—graphic, electronic, or mechanical, including photocopying, recording, taping, or information retrieval systems—without permission of the publisher.

The author and publisher make no warranty of any kind, expressed or implied, with regard to the programs and documentation contained in this book. The author and publisher shall not be liable in any event for incidental or consequential damages in connection with, or arising out of, the furnishing, performance, or use of the programs contained in this book.

Library of Congress Cataloging-in-Publication Data

Sobell, Mark G.

Building applications using a 4GL.

Includes index.

1. Electronic digital computers—Programming.
2. INFORMIX-4GL (Computer program language)
3. Programming languages (Electronic computers)
- I. Title.

QA76.S616727 1986 005.13'3 86-90094
ISBN 0-937613-00-2

BCDEFGHIJ-FB-8987

For more information on **INFORMIX-4GL** contact:

Informix Software, Inc.
4100 Bohannon Drive
Menlo Park, CA 94025
telephone 415/322-4100

Published by:
Sobell Associates
333 Cobalt Way, Suite 106
Sunnyvale, CA 94086
telephone 415/856-3460

PREFACE

Building Applications Using a 4GL is intended for people with some experience programming in a high-level language such as BASIC, COBOL, FORTRAN, Pascal, or C but is approachable by readers with minimal computer background. Experience with database management systems or 4GLs (fourth generation languages) is not required.

This book introduces you to the concepts of a 4GL so you can get to work building real-life applications quickly. It uses a hands-on approach, including many samples of actual 4GL code, to demonstrate concepts.

Builds a Practical Application

Building Applications Using a 4GL takes you through all the steps of building an actual 4GL application. It shows you how to design and code a typical application—a complete sales leads tracking system. Most of the examples come from the finished 4GL leads tracking system so, when you have finished reading the book, you should understand the ins and outs of the application.

Includes an Actual Demonstration

As an introduction to a 4GL, Chapter 1 gives you a demonstration of the sales leads tracking system that is the basis for the rest of the book. This is the kind of demonstration you would see if you asked a knowledgeable salesperson to show you a 4GL product. It goes into the features of the language without going into

a lot of depth. As you observe each of the features, the chapter also gives you a glimpse of the code that implements it.

Uses the Industry-Standard SQL Database Language

Because *Building Applications Using a 4GL* uses INFORMIX™-4GL for its examples, and because INFORMIX-4GL is based on the industry-standard SQL (Structured Query Language), all the portions of the book that discuss database manipulation use standard SQL terminology and statements. As you read this book, you not only learn general 4GL concepts, but you also get an introduction to the widely used SQL database language. Chapter 6 is dedicated to the SQL SELECT statement. It builds step by step from the simplest SELECT statement to complex multiple SELECTs that incorporate temporary tables, arithmetic, ordering, and grouping.

Step-by-Step Instructions on Getting Started

After the demonstration in Chapter 1, Chapter 2 starts on the ground floor by showing you how to design an application. In this chapter you will see how to analyze the requirements of your client or company and design an application that satisfies those requirements. Chapter 4 continues with instructions on how to convert the design into the code that specifies the structure of the database. This structure, called the schema, is the foundation for the application.

Menu Builder, Screen Handler, Report Writer

A complete chapter is devoted to each of these critical topics. Chapter 3 explains how to build a menu quickly that will be a prototype of your application and concludes with a structural prototype of the leads tracking system. Chapter 5 goes into detail about using screen forms and windows. It explains how to construct the forms and also looks at the code that you need to bring the forms and windows to life. The report writer is covered in Chapter 7. Again, this chapter starts with a simple report that prints out mailing labels and works its way up to a report that details the performance of salespeople over a six-month period.

Advanced Concepts

Chapter 8, the chapter on advanced concepts, explains how to set up a query by example, work with arrays, and trap interrupts. The chapter covers the powerful `DISPLAY ARRAY` and `INPUT ARRAY` statements in depth, showing you the details of scrolling on the screen and how to let the user manipulate information in the database with ease. The section on query by example explains the use of the `CONSTRUCT` and `PREPARE` statements in setting up a user-directed query of the database.

Acknowledgments

Without the help of all the people at Informix Software, Inc., this book would not have been possible. In my capacity as both an alpha and beta tester of `INFORMIX-4GL`, I received much help from Betty Chang and Dave Ellis in the Informix Software Engineering Department. Whenever a concept was not clear to me, I enlisted the support of “the Doctor” in the Tech Pubs Department: Dick Curtis. In the area of critical reviews, I have to thank Nick Baxter, Tommy Hawkins, Laura King, Gary McQue, Karen Rathjens, Roger Sippl, and, of course, Steve Marra for his help in actually putting the whole thing together. In addition, Roberta Harvey of RAH Consulting gave me valuable input.

Finally, I would like to thank Dr. Kathy Hemenway, without whose assistance the second edition of this book might never have come into being.

MGS

TABLE OF CONTENTS

INTRODUCTION 1

- Organization of This Book 5
- Objectives of This Book 6
- Audience 7
- Getting the Most from This Book 7

1 A PROBLEM AND ITS SOLUTION: A QUICK TOUR OF THE **leads** PROGRAM 9

- DEFINING THE PROBLEM 10
- DESIGNING A SOLUTION 10
- A DEMONSTRATION OF THE **leads** PROGRAM 11
 - About the Demonstration 12
 - Menus and On-line Help 12
 - Reports: Organizing the Output 15
 - Forms and Input 19
 - Query by Example 22
 - A Programmer's Perspective of **INFORMIX-4GL** 24
 - SELECT**: Retrieving Information from the Database 27
 - Coding a Form: Making the User's Job Easier 28
 - INPUT BY NAME**: Moving Data from the Screen to the Program 30
 - DISPLAY ARRAY**: Automatic Scrolling on Display 31
 - Manipulating the Database 32
- SUMMARY 37

2	DESIGNING THE APPLICATION	39
	SETTING UP THE MENU	40
	The leads Menu	40
	PLANNING THE REPORTS	42
	UNDERSTANDING TABLES, ROWS, COLUMNS, AND JOINS	47
	DESIGNING THE SCHEMA	51
	LAYING OUT THE FORMS	57
	SUMMARY	62
3	SETTING UP THE MENU	63
	A SAMPLE MENU	64
	SETTING UP THE leads MENU SYSTEM	66
	SUMMARY	70
4	BUILDING THE SCHEMA	71
	CREATING A TABLE	72
	Indexes	73
	CREATING THE leads SCHEMA	74
	GRANTING PRIVILEGES	77
	SUMMARY	78
5	USING FORMS	79
	CREATING THE FORM	80
	The Default Form	81
	Customizing a Form	82
	The Parts of a Form	83
	CONTROLLING THE FORM: THE INFORMIX-4GL CODE	85
	Variables, Declarations, Definitions, and Comments	85
	Working with the Form	89
	Getting the Data into the Table	91
	OPENING THE FORM IN A WINDOW	92
	The OPEN WINDOW Statement	93
	Multiple-Window Programs	97
	The Current Window	98
	SUMMARY	98

6	SQL AND SELECT	99
	SIMPLE SELECT STATEMENTS	100
	Using a Cursor	101
	The FOREACH Loop	103
	Defining Variables with LIKE	103
	Using a Record Variable	104
	JOINING TABLES	106
	ADVANCED SELECT STATEMENTS	112
	The GROUP BY Clause	114
	The ORDER BY Clause	117
	Subqueries and Temporary Tables	118
	SUMMARY	121
7	USING THE REPORT WRITER	123
	SETTING UP A REPORT	124
	The Report Driver	124
	The Report	126
	ANOTHER REPORT	128
	A MORE COMPLEX REPORT	132
	The Report Driver	133
	The Report	136
	SUMMARY	140
8	ADVANCED CONCEPTS AND FEATURES	143
	QUERY BY EXAMPLE	144
	ARRAYS	148
	Defining an Array	148
	Assigning Values to an Array	149
	Setting Up the Form	150
	The BEFORE FIELD and AFTER FIELD Clauses	151
	The DISPLAY ARRAY Statement	153
	Putting the Data into the Table	156
	INPUT ARRAY	157
	TRAPPING INTERRUPTS	167
	SUMMARY	168

Appendix A LISTING OF THE leads PROGRAM 171

PROGRAMS 172

create.4gl 172

menu.4gl 174

FUNCTIONS 176

contact.4gl 176

decide.4gl 179

globals.4gl 182

product.4gl 183

prospect.4gl 185

qcontact.4gl 186

sperson.4gl 189

wwait.4gl 190

FORMS 191

f_contact.per 191

f_decide1.per 191

f_decide2.per 192

f_decide3.per 192

f_empnum.per 193

f_listprod.per 193

f_pdesc.per 194

f_prospect.per 194

f_qcontact.per 195

f_sale.per 196

f_sperson.per 197

REPORT DRIVERS 198

rd_follow.4gl 198

rd_label.4gl 199

rd_letter.4gl 199

rd_sales.4gl 200

rd_sperson.4gl 202

REPORTS 203

r_follow.4gl 203

r_label.4gl 204

r_letter.4gl 205

r_sales.4gl 207

r_sperson.4gl 209

SIMPLIFIED PROGRAMS AND FORMS 210

f_scontact.per 210

f_sproduct.per 210

s_contact.4gl 211

s_medium.4gl 213

s_product.4gl 214

Appendix B ABOUT NULLS 217

**Appendix C SETTING UP THE leads DATABASE
AND COMPILING PROGRAMS 221**

SETTING UP THE leads DATABASE 222

COMPILING AN INFORMIX-4GL PROGRAM 223

BUILDING A FORM 225

FILENAME EXTENSIONS 226

Appendix D THE leads DATABASE 227

The contact Table 228

The product Table 232

The prospect Table 232

The sale Table 233

The sperson Table 234

**Appendix E THE STRUCTURE OF THE
leads APPLICATION 237**

INDEX 239

INTRODUCTION

Advances in programming languages over the past 30 years can be broken into roughly four generations. Each generation has made it easier and more practical to implement sophisticated applications.

Writing code used to be the lion's share of solving a problem. Machine code, strings of 1s and 0s, was the first generation. It was tedious to write *any* program using machine language.

The second and third generations improved things considerably. An insurance application that was nearly impossible to implement in assembly language (second generation) took only a few years to write in COBOL (third generation). Structured languages such as C and Pascal (still the third generation) reduced development time even more. Database tools such as database management systems, report writers, and screen handlers made it possible to complete the same task in less than a year.

Today, fourth generation languages (4GLs) are cutting development time even further. Increasingly, application builders are taking advantage of the power and flexibility of 4GLs to implement solutions and bring up new applications faster than ever before. Using high-level, intuitive programming constructs, 4GLs automate much of the repetitive detail work, so applications written in 4GLs are also easy to maintain and enhance.

4GLs are powerful because they are largely *nonprocedural*—you tell the computer *what* you want, instead of telling it *how* to get it. Whereas with procedural, third generation languages you

had to give detailed instructions describing how something should be accomplished, with 4GLs you specify the result. Using a 4GL is like telling a cook what you want to eat without providing the recipe: you specify what you want (a double-crust apple pie, for example) and leave the specific procedures up to the cook. Although developing a detailed recipe is time-consuming and requires a knowledge of cooking, describing the result requires only an understanding of the end product. Similarly, nonprocedural languages enable the application builder to focus on the result rather than on the details of the programming language.

Unlike skilled cooks, however, nonprocedural languages can be inflexible. Although you can tell a cook to make a light, moist crust and he or she will figure out how to do it, nonprocedural languages sometimes do not support unanticipated features in an application. Consequently, they can severely limit the application builder's ability to customize a program. To provide the flexibility to handle customizations that were not anticipated by the languages' designers, the best 4GLs include some procedural operations.

Writing the code that implements a solution is only part of solving a problem. In the past, once a system of assembly language programs was "complete," it took a score of programmers to maintain the code, assuming no significant changes were required. Maintaining a complex COBOL program was a full-time job for at least one programmer. Applications written using structured languages took less time to maintain, and now programs written using 4GLs are even easier to keep current and enhance. Because they require less code and because the code is easy to read and follow, it is simple to maintain complicated applications that would have been maintenance nightmares if written in third generation languages. Also, programs written in conventional languages were often difficult to comprehend after you had spent some time away from them. Programs written with high-level, fourth generation constructs are much easier to grasp.

In addition to speeding software development and easing maintenance, 4GLs have changed the nature of the development process. Through all the generations of languages, the

identification of the problem and the design of the solution (the systems analysis phase) have always been critical to the success of an application. Database systems and 4GLs have changed the focus of this phase, as well as its relationship to the subsequent implementation phase. Before database systems and 4GLs, evaluating the feasibility of implementing proposed solutions was a big part of the design process. Because 4GLs alleviate most implementation problems, the analyst can now focus on analyzing the problem, with the assurance that implementation will quickly follow. As the analyst, you can work at a much higher level, more in line with the problems the application is intended to solve.

4GLs relieve the designer of much of the concern about the user interface because they provide high-level tools that make building sophisticated, easy-to-use user interfaces simple. One of the hallmark features of a 4GL is a well-integrated set of user interface tools that includes a screen handler, menu manager, and windowing system. Using these tools, it is easy to design and develop sophisticated, screen-oriented user interfaces. Windows allow you to take maximum advantage of space on the screen: you can present many more fields, prompts, or messages on a single screen without making it look busy or cluttered. Windows keep the screen simple and free of clutter by temporarily usurping screen space, providing a change of context and change of focus for the user. They enable you, the designer, to group fields in meaningful ways and to channel the user's attention to sets of fields. Because windows provide an easy way to present descriptions of alternative choices for a field, users can select information from a window rather than having to memorize and enter abbreviations or codes in fields.

4GL database tools make the design phase more productive by enabling you to solve problems through exploratory development work. With the introduction of database tools, you no longer had to wait until a system was complete before you could see what needed to be changed. And even if the system was complete, you could make changes without rewriting it entirely. With 4GLs it has become practical to develop prototypes as a routine part of systems analysis and to reuse the prototype code in the final development. When you develop a 4GL prototype as a systems

analysis tool, you have already developed a program structure that you can use when you go into the implementation phase.

4GLs give you, the application builder, all the tools you need to speed prototyping and application development:

- Integrated, sophisticated, development environments
- Flexible data access methods and high-level end-user interfaces that allow you to concentrate on the application at hand
- Menu systems that allow a user to go from screen to screen with ease and confidence
- Screen handlers (forms) that allow you to code an easy-to-use user interface quickly
- Pop-up windows that allow a user to work with detailed information in a form or to change contexts temporarily
- Data validation tests you can easily incorporate into a program and into the data dictionary
- Nonprocedural report writers that automatically handle the formatting and arithmetic aspects of a report while you concentrate on the contents
- Help systems that are easy to build into an application and assist users in getting familiar with new applications and in finding new ways of using existing applications

Until the advent of 4GLs, these features were not all available in any one language. Each was available, but coordinating the different tools was difficult to impossible. 4GLs give you all the tools you need in one seamless language. Equipped with a 4GL you, the systems analyst and programmer, can complete the design and implementation phases in record time. With your attention no longer focused on low-level implementation details, as an analyst you can focus your attention on the problem your application is intended to solve. You can spend your time searching for the optimal solution and writing prototype code. As an implementor your job is also much easier: you can rely on the built-in menu handler, screen handler, windowing system, and report writer to do much of the work for you.

With the advent of 4GLs, the design portion of a project has not only become a larger portion of most projects proportionately,

but it has also enabled programs to rise from the mediocrity of hard-to-use coding nightmares to the superiority of easy-to-use, critical applications. Instead of designing an application merely to fill the immediate perceived needs of an organization, you can, with a little extra investigation, design an application that will allow the organization to take advantage of unthought-of possibilities that stem from the quick and accurate presentation of information that is organized in new and useful ways.

Organization of This Book

In order to develop a 4GL application, you must specify how the computer is to collect and store data and then how it is to summarize the data and create reports filled with useful information. This book identifies the steps of the development process and details the most useful features of a 4GL. It draws its examples from one of the most powerful and flexible fourth generation languages available: Informix Software's **INFORMIX-4GL**.

The first chapter sketches the development of a complete application, focusing on the user interface and the corresponding **INFORMIX-4GL** features. This chapter introduces the **leads** application program that is used throughout the book. The **leads** program solves the problems a sales staff meets trying to keep track of customers, sales leads, and scheduled contacts. It also solves one of management's problems by generating reports on the comparative effectiveness of each salesperson over time.

Chapter 2 details a practical design philosophy that will allow you to create applications that solve your needs or those of your corporation or clients. It presents a structured approach to using a 4GL to solve a problem. This chapter explains how to design the program output—the reports—and then how to use these designs to determine what data the database must store. It concludes with a discussion of the screen forms the user will need in order to work with the data in the database.

Chapter 3 discusses menus. It explains how to take advantage of **INFORMIX-4GL**'s menu capabilities to develop prototype systems that can form the foundation of a complete application.

The schema is the outline for the data the database stores.

Chapter 4 covers the coding of the schema that was developed in the second chapter.

Although you will usually *design* reports before forms, it is a good idea to *implement* forms first because you can use forms to enter data (including test data) whereas a report cannot work until after you enter data. Chapter 5 explores forms and discusses some of the implementation issues surrounding data input validation. It also explains how to use forms with windows.

Before you can code the application itself, you need to know how to extract information from the database. Extracting this information is the job of the SELECT statement. The SELECT statement is the basis for SQL (the industry-standard Structured Query Language), the database manipulation portion of INFORMIX-4GL. Chapter 6 discusses SELECT statements of varying complexity, along with the uses of different types of variables.

Chapter 7 explains what a report writer can do, detailing how to code a report. It explains some of the important concepts of a report writer, including grouping, page breaks, and formatting.

Chapter 8 describes some of the more advanced coding techniques that were used in preparing the **leads** program and goes into the program logic. This chapter includes a discussion of INFORMIX-4GL's powerful CONSTRUCT and PREPARE statements and when and how you can use them to allow the user to perform ad hoc queries. It also covers the INPUT ARRAY and DISPLAY ARRAY statements that implement scrolling on the screen and permit the user to manipulate data in an array.

Appendix A is a complete listing of the source code for the **leads** program. Appendix B describes the properties and uses of nulls. Appendix C explains how to compile an INFORMIX-4GL program and how to set up the **leads** demonstration database and program. Appendix D is a complete list of the data from the **leads** database. Appendix E contains an illustration that shows the structure of the **leads** application.

Objectives of This Book

After examining the development process and the code that creates the application, you will be able to:

- Modify or extend this book's example programs to meet your needs
- Create a new **INFORMIX-4GL** application program
- Modify the application development process described in this book to suit your particular 4GL
- Build on your understanding of 4GLs using the documentation that comes with your particular 4GL to take full advantage of the language's features

Audience

In order to take maximum advantage of this book, it is helpful if you have had some experience programming in BASIC, FORTRAN, COBOL, C, Pascal, or some other high-level language or with Lotus 1-2-3 (or other) macro commands. In addition, experience in database programming will give you a head start on some of the concepts this book presents, but it is not required.

Getting the Most from This Book

Application programs are dynamic, functioning, interactive entities, best seen and understood through use. If you have a computer that runs **INFORMIX-4GL**, you can install the **leads** demonstration application on your system by following the instructions in Appendix C (the **leads** database and program are included as part of **INFORMIX-4GL**). Experiment with each feature as you proceed through the book. If you cannot run **leads**, refer often to Appendix A (the **leads** program listing) and Appendix D (the **leads** data) as you read the book to familiarize yourself with the program and data.

1

A PROBLEM AND ITS SOLUTION: A QUICK TOUR OF THE leads PROGRAM

This chapter demonstrates the powers of a fourth generation language. It starts by defining a problem and sketching out a solution. Then it presents an overview of the finished solution as implemented in **INFORMIX-4GL**. Not only does this chapter show what the user sees, but it also gives you glimpses of the code that creates the menus, forms, and reports. It is an introduction to a 4GL.

DEFINING THE PROBLEM

Just what is the problem you need to solve? Sometimes the answer is easy. “I have all my data coming from one place and need these three reports once a month, along with quarterly and yearly summaries.” At other times the input data is coming from different places and exists in different formats, the understanding levels of people using the system vary widely, the reports must correlate diverse groups of data, and the user is demanding complete flexibility for ad hoc queries.

Before you can design a solution, you have to specify completely the problem you are trying to solve. Once you have defined the problem and designed a solution, you can augment the solution so it not only solves the problem at hand but also takes further advantage of the available data and the technology of a 4GL to bring additional benefits to the user.

This book presents the case of the Associated Binder Company (ABC). ABC’s problem is that it has grown to a size where its manual system of sales leads tracking no longer works. Leads from ads take more than a month to filter down to the salespeople who follow up on them. Second and subsequent contacts with all but the largest potential customers are unheard of. Reports to management are late, inaccurate, and sketchy at best.

DESIGNING A SOLUTION

Often when you are starting to design a solution using a 4GL, it is useful to think of turning the computer into a machine that has a very, very, high-level programming capability. With this theoretical programming language, the user needs only to push a single button to complete a task. Your job as a designer is to label the buttons.

To solve ABC’s problems listed above, you will need buttons that enable the user to:

- Enter information on salespeople and products
- Enter a new lead: both information on the prospect (for example, name, address, phone number) and on the contact (for example, date, type of contact, notes about the contact)
- Enter the information about a second or subsequent contact with a prospect
- Print a report for each salesperson on contacts that need to be followed up in the next week
- Display a report on sales by salesperson
- Update information about a prospect
- Update information about a contact with a prospect
- Make changes to the list of salespeople
- Make changes to the list of products

While you are labeling buttons, you will usually come up with some that do not specifically solve the problems at hand but do present useful information. In the case of ABC, the key to providing additional useful information is taking the data that the system gathers one step further. Following are some more possible button labels:

- Print follow-up letters in response to contacts
- Print mailing labels for all prospects

The rest of this chapter is a demonstration of the finished program. Look for the labels mentioned above as menu selections in the following pages. The balance of this book, after this chapter, is devoted to explaining how to design and code the program that will implement this solution.

A DEMONSTRATION OF THE **leads** PROGRAM

This section presents an overview of the **leads** application program that was implemented to solve the problems of the Associated Binder Company. It directs your attention to how you can

1. A PROBLEM AND ITS SOLUTION

use **INFORMIX-4GL**'s features to solve problems. Although it is technical, this chapter does not attempt to explain each of the features in comprehensive detail. The in-depth explanations are left for the rest of the book.

About the Demonstration

You do not need to have a computer or a copy of **INFORMIX-4GL** to read this section, but if you have a copy of **INFORMIX-4GL** and you want to install the **leads** database and program on your computer, refer to Appendix C for installation instructions. (The **leads** application is included with **INFORMIX-4GL** when you buy it.) Appendix A contains listings of the source code for the entire **leads** application program.

The demonstration starts with a look at the first screen you see when you run **leads**, the **Top-Level** menu, followed by the **INFORMIX-4GL** code that implements this menu. When you run the demonstration program, the **Top-Level** menu allows you to choose what you want to do first.

Menus and On-line Help

Because **INFORMIX-4GL**'s statements are so powerful, they allow you to perform a lot of work with just a few lines of code. Figure 1-1 shows the first screen you see when you call up the **leads** program—it is implemented by the code shown in Figure 1-2.

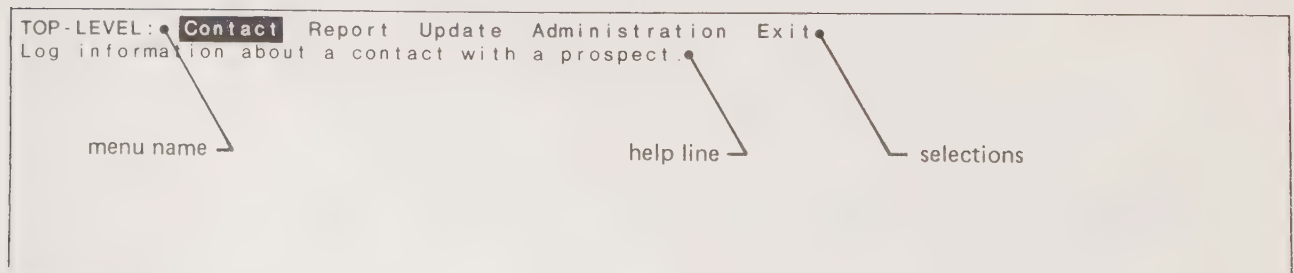


Figure 1-1: The **Top-Level** Menu on the Screen

With the menu displayed on the screen, if you know which menu selection you want, you can simply press the key that corresponds to the first letter of that selection. If you want to see a brief description of each selection, you can repeatedly press the SPACE bar. Each time you press the SPACE bar, the highlight at the top of the screen will move one selection to the right and the help line (the second line on the screen) will change to reflect the

```

DATABASE leads
GLOBALS "globals.4gl"

MAIN
{
The menu program initially displays the Top-Level menu.
Depending on the user's selection, it then either displays a
submenu or calls a function or report driver. When a submenu
is called, it in turn invokes a submenu or calls a function or
report driver, and so forth.
}

MENU "TOP-LEVEL"
{
COMMAND "Contact" "Log information about a contact with a prospect."
.
.
COMMAND "Report" "Run a report."
MENU "REPORT"
COMMAND "Mailing-Labels" "Generate labels for a prospects mailing."
CALL rd_label()
EXIT MENU
COMMAND "Sales" "Run a report on sales by salesperson by month."
CALL rd_sales()
.
.
.
END MENU
.
.
COMMAND "Update" "Update information about a contact or prospect."
.
.
COMMAND "Administration" "Change information about a product or salesperson."
.
.
COMMAND "Exit" "Return to the operating system."
EXIT MENU
END MENU

```

Annotations in the diagram:

- menu name**: points to "TOP-LEVEL"
- help line**: points to "Log information about a contact with a prospect."
- selections**: points to the list of command names ("Contact", "Report", "Mailing-Labels", "Sales", "Update", "Administration", "Exit")
- selections**: points to the list of descriptions ("Log information about a contact with a prospect.", "Run a report.", "Generate labels for a prospects mailing.", "Run a report on sales by salesperson by month.", "Update information about a contact or prospect.", "Change information about a product or salesperson.", "Return to the operating system.")

Figure 1-2: Part of the Code for the **Top-Level Menu**

1. A PROBLEM AND ITS SOLUTION

newly highlighted selection. When the highlight is on the selection you want, press RETURN.

This kind of menu allows both experienced and inexperienced users to use the finished application program at top speed, obtaining assistance in the form of a help line only as required. If you anticipate that users will need more assistance than help lines can provide, INFORMIX-4GL provides on-line, context-sensitive, custom program documentation (help screens). If you have made provision in your INFORMIX-4GL program for on-line documentation, a user need only press a predefined help key at any time while running the program and INFORMIX-4GL will display the documentation text you have specified for that routine. (This feature is not shown.)

IDENTIFIER NAMES

Prefix	Identifier Type
c__	cursor
f__	form
i__	table index
pa__	program array variable
pr__	program record variable
rr__	program report record
r__	report
rd__	report driver
s__	simplified program
sr__	screen record
w__	window

USE OF CASE

Uppercase	Lowercase
keywords	programmer-created identifiers
menu names	menu selection labels

Figure 1-3: Naming Conventions Used in the **leads** Application Program

INFORMIX-4GL is not case-sensitive. In order to enhance the readability of the examples in this book, the examples show keywords in uppercase letters and programmer-created variables and functions in lowercase letters. You can use uppercase or lowercase as you please. The conventions used in naming variables, forms, report drivers, reports, and other identifiers in the **leads** application are listed in Figure 1-3.

Reports: Organizing the Output

One of the most important features of a 4GL is a well-integrated report writing facility, frequently referred to as a *report writer*. A report writer gives you extensive control over the format of information a program produces. Although you can do a lot of formatting work without it, a report writer frequently makes your job easier and, in some cases, allows you to do things you could not do otherwise. The output from the report writer can go to a printer to create a conventional report such as a sales report, it can be directed into an operating system file for storage or further processing, or it can go directly to the screen.

When you choose **Administration** from the **Top-Level** menu and then select **Salesperson** from the **Administration** menu, the **leads** program displays the **Salesperson** menu (Figure 1-4).



```
SALESPERSON:  Add  Update  Delete  List  Exit
Add a new salesperson.
```

Figure 1-4: The **Salesperson** Menu

If you choose **List** from the **Salesperson** menu, **leads** displays the output of the **r__sperson** (short for report: salesperson) report (Figure 1-5).

1. A PROBLEM AND ITS SOLUTION

LIST OF CODES AND SALESPeOPLE

Code	Name
126	Bell, Jerome
127	Fuller, Robert
125	Greenberg, Marlene
128	Kimball, Molly
129	Richardson, Isabelle
130	Wieseman, Frank

Press any key to continue. ■

Figure 1-5: Output from the `r__sperson` Report

The parts of the report code that control the format of the report output are the `OUTPUT` and `FORMAT` sections. Figure 1-6 shows these parts of the report.

The `OUTPUT` Section

The `OUTPUT` section of a report controls the margins, page length, and destination of the report output. In the example, the report output goes to the screen because that is the default destination when you do not specify a destination in the `OUTPUT` section. Because the report is destined for the screen, the top and bottom margins are set to 0. The page length is set to 22 so one page of the report output fits exactly on a 24-line screen together with a prompt line and a blank line.


```
OUTPUT
  TOP MARGIN 0
  BOTTOM MARGIN 0
  PAGE LENGTH 22

FORMAT
  PAGE HEADER
    CLEAR SCREEN
    PRINT "LIST OF CODES AND SALESPEOPLE"
    SKIP 1 LINE
    PRINT "Code", COLUMN 15, "Name"
    SKIP 1 LINE

  PAGE TRAILER
    CALL wwait()

  ON EVERY ROW
    PRINT empnum, COLUMN 15, lnm CLIPPED, ", ", fnm CLIPPED

END REPORT
```

Figure 1-6: The OUTPUT and FORMAT Sections of the **r_sperson** Report

The FORMAT Section

The FORMAT section of a report controls what gets printed and what format it gets printed in. Refer to Figure 1-6. The FORMAT section of the **r_sperson** report has three subsections: PAGE HEADER, PAGE TRAILER, and ON EVERY ROW. **INFORMIX-4GL** executes the statements in the PAGE HEADER subsection at the top of each page of the report. The example executes a CLEAR SCREEN statement to start each new page (screen). Then it uses PRINT to display a report header and column headers. The PAGE TRAILER subsection executes a CALL statement after the report has filled the screen with information. The CALL statement executes the user-defined **wwait** function (subroutine), which prompts the user and waits until the user presses a key before it continues. In the example, **wwait** holds the information on the screen until the user presses a key. Then it allows the report to continue.

The PRINT statement in the ON EVERY ROW subsection displays the substance of the report output. For each row the report works on, this statement displays the values of three variables: **empnum** (employee number), **lnm** (last name), and **fnm** (first

1. A PROBLEM AND ITS SOLUTION

name). The employee number starts at the default left margin (column 5), and the name starts in column 20 (because of the COLUMN 15 clause and the default margin). The last and first names are separated by a comma. The CLIPPED clauses eliminate all trailing blanks when the first and last names are printed, ensuring that the comma appears immediately after the last name and speeding the display of the report by not printing blanks.

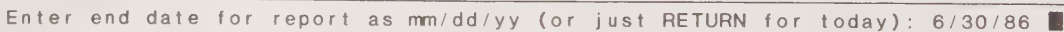
This is a simple report. You can also perform many types of formatting, grouping, sorting, totaling, and subtotalling within a report. When you choose **Report** from the **Top-Level** menu, the leads program displays the **Report** menu (Figure 1-7).



```
REPORT:  Mailing-labels  Sales  Follow-up  Letters  Exit
Generate labels for a prospects mailing.
```

Figure 1-7: The Report Menu

If you choose **Sales** from this menu, **INFORMIX-4GL** will prompt you for the ending date of the report (Figure 1-8) and then display the report output showing the sales figures, by employee, for the six months ending on the date you specified (Figure 1-9).



```
Enter end date for report as mm/dd/yy (or just RETURN for today): 6/30/86 █
```

Figure 1-8: The Sales Report Prompting for a Date

Refer to **r_sperson**, **r_sales**, **rd_sperson**, and **rd_sales** in Appendix A if you want to see the code for these reports and to Chapter 7 for more information on reports.

S A L E S							
for the period Jan. 01, 1986 - Jun. 30, 1986							
by salesperson							
Salesperson	Jan	Feb	Mar	Apr	May	Jun	Total
J Bell	31	0	280	435	190	95	1,031
R Fuller	12	181	112	45	0	49	399
M Greenberg	372	130	0	170	213	723	1,608
Total	415	311	392	650	403	867	3,038
Press any key to continue. ■							

Figure 1-9: The Sales Report Output

Forms and Input

In the past, one of the most labor-intensive aspects of coding an application was creating the screens that allowed the user to interact with the program. With the advent of screen handling programs, this task became easier, but there was still no good way to interface the screen handling routines with the other parts of a program. One of the earmarks of a 4GL is the ability to embed a screen handling routine in the middle of a program or, just as easily, embed pieces of a program in the middle of a screen handling routine.

Returning to the needs of the Associated Binder Company, when a salesperson talks with a prospect, the information about

that contact needs to be entered into the database. (In the **leads** program, a *contact* is an event—for example a meeting, phone call, or letter—and a *prospect* is a prospective customer.) When the salesperson chooses **Contact** from the **Top-Level** menu, the **leads** program, after asking the salesperson for any necessary information on the prospect, calls the **contact** function. The **contact** function works together with the **f_contact** form to help a user enter information about a contact with a prospect. To the extent possible, the form and the function verify the data the user enters and, for some fields, suggest possible entries. The user can override any of the suggestions (defaults) by entering a value on top of the suggested value. Figure 1-10 shows a simplified version of the **f_contact** form, **f_scontact**, as a user is filling it in.

The simplified version of the **contact** function, **s_contact**, uses windows to enable the user to enter a lot of information on a single screen. The top part of Figure 1-10 shows the **f_scontact** form as the user is filling it in. Once the user puts S in the type field, indicating that the contact was a sale, the **w_sale** window pops up with fields to fill in data about the sale (shown in the middle screen in Figure 1-10). Because these fields were included in the **w_sale** window rather than on the original **f_scontact** form, the **f_scontact** form is free of the clutter of these fields, which are used only for some contacts. Aside from providing a conceptual grouping for the sale fields, when the window pops up it provides a temporary change of context, focusing the user's attention on the fields at hand.

In the middle screen in Figure 1-10, the user has entered a 0 in the product code field to request assistance in selecting a product code. The program opens the **w_pdesc** window, which displays a list of product descriptions (lower screen in Figure 1-10). This window pops up to provide help if the user requests it—in effect, it provides a menu of selections for the user to choose from. This feature enables novices, or other users who are not familiar with product codes, to use the form effectively without learning the codes. The user can use the **ARROW** keys to move the highlight up and down the list of products—**INFORMIX-4GL** automatically scrolls the list as needed. As the message at the top of the screen indicates, the user can press **ESCAPE** to choose the product that

Enter C(complaint) I(request info) Q(inquiry) S(sale)

CONTACT INFORMATION:

Date 04/28/1986
 Salesperson 125
 Next 05/12/1986
 Next salesperson 125
 Comments He was satisfied with the replacement
 Type █



CONTACT INFORMATION:

Enter 0 to display and choose from a list of values.

Ne SALES INFORMATION:
 Product code 0 █
 Quantity sold
 Discount



CONTACT INFORMATION

Ne

SAL

Use ARROW keys to move highlight, ESCAPE to make selection.

- 1. 0" D-ring (special order)
- 1. 0" regular binder
- 1. 5" D-ring binder
- 2. 0" D-ring binder
- 2. 5" D-ring binder

Figure 1-10: Filling In the **f_scontact** Form on the Screen

the highlight is on. When the user chooses a product description, the program puts the corresponding code in the product code field and closes the window. When the `w_pdesc` window is closed, the underlying `w_sale` window is restored, and the user can finish filling in information about the sale.

Query by Example

But what if the salesperson who is entering the contact information makes a mistake? Or wants to add something to the comments field as an afterthought? What is the easiest way for the salesperson to recall the contact information to the screen for updating? A query by example is the answer. A query by example makes it possible for a nontechnical user to perform sophisticated, nonstandard queries.

When you choose **Update** from the **Top-Level** menu, you will see the menu shown in Figure 1-11.

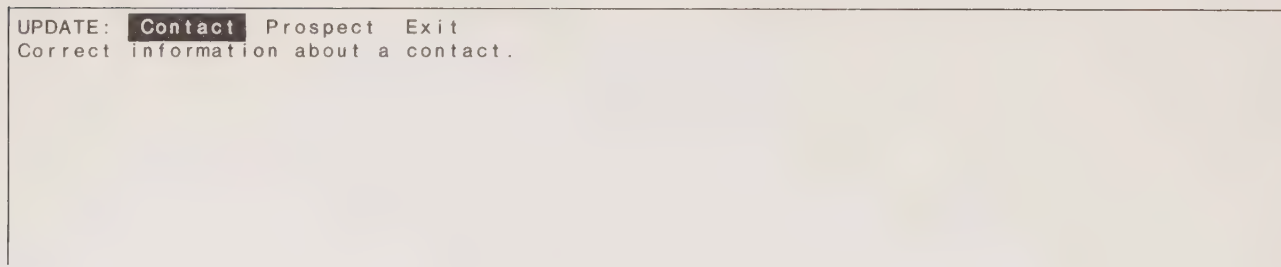


Figure 1-11: The Update Menu

If you choose **Contact** from this menu, the `leads` program calls the `qcontact` function. The `qcontact` function displays the `f_qcontact` form, a blank form similar to the `f_contact` form, and its associated windows and lets you fill in any number of fields to identify the contact that needs to be updated (Figure 1-12).

UPDATE CONTACT INFORMATION

To display the row for updating, fill in some fields, then press ESCAPE.
Once the row is displayed, update the appropriate fields, then press ESCAPE.

CONTACT INFORMATION:

Date [<1/1/86]
 Salesperson []
 Next []
 Next salesperson []
 Comments []
 Type []

PROSPECT INFORMATION:

Last name [McCaffrey]
 Company name []

SALES INFORMATION:

Product code []
 Quantity sold []
 Discount []

Figure 1-12: The **f__qcontact** Form, Used for a Query by Example

In a query by example, you can use relational operators with dates and numbers (<1/1/86 will find all contacts before the start of the new year). When you fill in more than one field, the query will return only contacts that satisfy all the criteria. The example in Figure 1-12 will find all contacts with a prospect named McCaffrey that occurred before the end of 1985.

When the user presses **ESCAPE**, the **qcontact** function takes the results from the query by example and displays them on the same form that was used to perform the query. As **qcontact** displays each contact that satisfies the terms of the query, the function prompts the user to see if it is the one that needs to be updated (Figure 1-13). The message at the top of the form provides instructions on how to update fields on the form.

1. A PROBLEM AND ITS SOLUTION

```
RETURN for next row; u to update this row.

UPDATE CONTACT INFORMATION
To display the row for updating, fill in some fields, then press ESCAPE.
Once the row is displayed, update the appropriate fields, then press ESCAPE.

CONTACT INFORMATION:
      Date [ 12/06/1985 ]
      Salesperson [ 126 ]
      Next [ 01/15/1986 ]
      Next salesperson [ 126 ]
      Comments [ They might be a retail outlet ]
      Type [ S ]

PROSPECT INFORMATION:
      Last name [ McCaffrey ]
      Company name [ HEALTHWAY NATURAL FOODS ]

SALES INFORMATION:
      Product code [ 20d ]
      Quantity sold [ 15 ]
      Discount [ 10 ]
```

Figure 1-13: The `qcontact` Function Returning Information and Prompting the User

A Programmer's Perspective of INFORMIX-4GL

Before continuing with the demonstration, this section presents a brief overview of the structure and terminology of **INFORMIX-4GL**. This overview should make the terminology and program logic of the balance of the demonstration more understandable.

As an **INFORMIX-4GL** programmer, you can make use of many powerful statements to control the flow and manipulation of data. The resources you can send data to and accept data from are the database stored in the computer's memory and the user's terminal. In addition, you can send report output to the printer, the terminal screen, and operating system files. Refer to Figure 1-14.

When you use **INFORMIX-4GL**, you will be working with a relational database. A relational database stores data in structures called *tables* that are composed of *columns* and *rows*. Each table holds data pertaining to one subject (for example, prospect or sales data); each column stores a specific kind of data (for example, name or date of sale); and each row holds one complete set of whatever data the table stores (all the information about one

prospect or one sale). Chapter 2 discusses database structure and terminology in more detail.

When you work with **INFORMIX-4GL**, you can use different types of variables to store data as you move and manipulate it. The variables are just like those in most high-level languages. In addition to ordinary program variables, there are records of variables you can set up to correspond to database tables and arrays of variables that can hold many records. Screen records that contain variables that correspond to fields on the user's terminal screen are also available. Each type of variable has its place and use as you will see throughout this book.

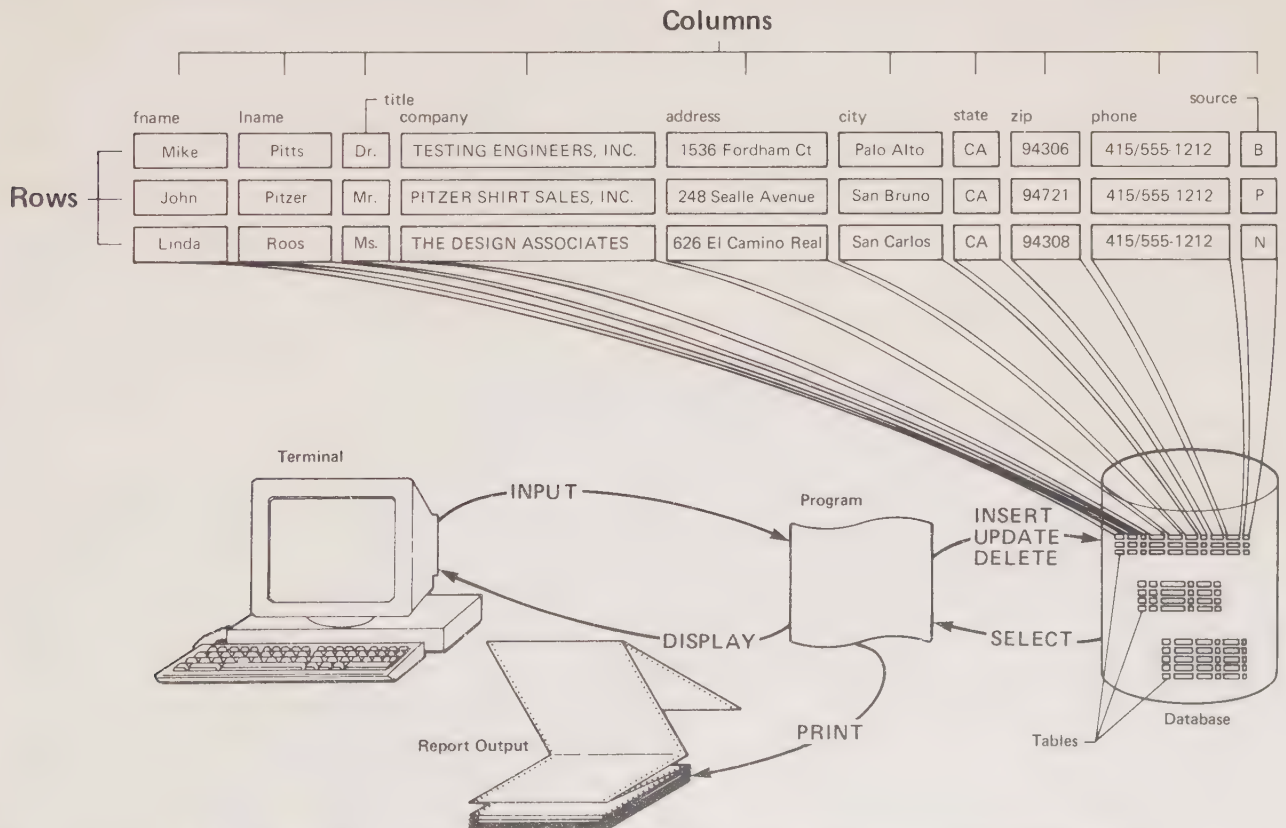


Figure 1-14: A Programmer's Perspective of **INFORMIX-4GL** with a Detail of the **prospect** Table

Figure 1-14 shows some of the important statements you can use to manipulate data and to move it between the screen, program variables, and the database. These statements, along with their functions, are listed below.

INPUT	Accepts data from the user via the terminal keyboard and puts it in variables. When combined with some of its clauses, INPUT can appear as INPUT BY NAME, INPUT WITHOUT DEFAULTS, and INPUT ARRAY.
DISPLAY	Takes data from variables and displays it on the terminal screen. When combined with some of its clauses, DISPLAY can appear as DISPLAY BY NAME and DISPLAY ARRAY.
SELECT	Fetches information from one or more tables in a database and puts it in variables.
PRINT	Takes data from variables and sends it to the terminal, printer, or operating system file. This statement is used in reports only.
INSERT	Takes data stored in variables and adds it to a table in a database.
UPDATE	Takes data stored in variables and uses it to update one or more rows of a table in a database.
DELETE	Removes one or more rows from a table in a database.

The purpose of the main program of the **leads** application, **menu**, is to display the menu and call the appropriate function depending on the user's menu selection. A typical function retrieves data from the database using SELECT and uses DISPLAY to display a form and data on the screen. Then INPUT accepts data from the user and INSERT, UPDATE, or DELETE makes corresponding changes to the database. If the function is a report driver, it retrieves data using SELECT and starts a report. The report, in turn, uses PRINT to send the data to the terminal, printer, or operating system file. Appendix E shows report drivers and reports as well as other functions and forms used by the **leads** application.

SELECT: Retrieving Information from the Database

The purpose of storing data in a database is to be able to retrieve it in the form of useful information. In an SQL-based 4GL such as INFORMIX-4GL, it is the job of the **SELECT** statement to retrieve information from the database in the form that is most useful to you. The **SELECT** statements can be quite simple or very complex. They can draw data from a single table or from several tables. Chapter 6 goes into greater detail about SQL and this important statement.

The report output shown in Figure 1-5 was gathered from the database by the **SELECT** statement shown in Figure 1-15.

```

SELECT      empnum,  } columns
            lname,   }
            fname    }
      INTO   pr_sperson.empnum, } elements of a record variable
            pr_sperson.lname,  }
            pr_sperson.fname   }
      FROM   sperson
      ORDER BY  lname,
               fname

```



Figure 1-15: A **SELECT** Statement from **rd_sperson**

The words following the keyword **SELECT** are the names of the columns that **SELECT** will fetch values from. The **SELECT** in Figure 1-15 fetches values from the **empnum**, **lname**, and **fname** columns. The **INTO** clause gives **SELECT** somewhere to put the values that it fetches. The **pr_sperson.empnum**, **pr_sperson.lname**, and **pr_sperson.fname** elements of the **pr_sperson** record correspond to the appropriate columns in the **sperson** table—the table **SELECT** is fetching values from. As **SELECT** fetches values from the **sperson** table, the **ORDER BY** clause ensures that the rows are returned in order, sorted by the **lname** and **fname** columns.

This **SELECT** fetches all rows from the table because it has no **WHERE** clause to qualify rows. An example of a **WHERE** clause that qualifies rows is **WHERE empnum > 900**. This **WHERE** clause would cause **SELECT** to fetch only rows containing employees with an employee number greater than 900.

1. A PROBLEM AND ITS SOLUTION

When you choose **Sales** from the **Report** menu, the **leads** program calls the **rd_sales** report driver to run the **r_sales** report (refer back to Figure 1-9). This report uses the **SELECT** statement shown in Figure 1-16 to generate information.

```
SELECT      sale.quantity * product.price * (1 - sale.discount/100) tot,
            MONTH(contact.cdate) mon,
            contact.empnum num
FROM        sale,
            contact,
            product
WHERE       sale.cnum = contact.cnum
            AND sale.pcode = product.pcode
            AND contact.cdate BETWEEN sdate AND edate
INTO TEMP  hold
```

Figure 1-16: A More Complex **SELECT** Statement from **rd_sales**

The **SELECT** in Figure 1-16 uses arithmetic operators and a date function (**MONTH**) as it joins rows from three tables (more on joins in the next chapter). The **WHERE** clause establishes a relationship between the tables and also rejects rows with a contact date (**cdate**) that is not between the report's start date and end date. The results of this **SELECT** go directly into the temporary table named **hold** for further processing.

Coding a Form: Making the User's Job Easier

Figure 1-17 shows the **INFORMIX-4GL** code for the form that the **s_contact** program uses (refer back to Figure 1-10).

The code for the form shown in Figure 1-17 has five sections: **DATABASE**, **SCREEN**, **TABLES**, **ATTRIBUTES**, and **INSTRUCTIONS**. The **DATABASE** section declares which database the form will work with, and the **TABLES** section declares which database tables the form will work with.

The **SCREEN** section is an image of what will appear on the screen; everything outside the square brackets (**[]**) will appear on the screen exactly as it appears in the code for the form. The characters within the brackets are called *field tags* and do not appear on the form when it is displayed on the screen. The

DATABASE leads
SCREEN

```
CONTACT INFORMATION
    Date: c01
    Salesperson: c02
    Next: c03
    Next salesperson: c04
    Comments: c05
    Type: c
```

field tags

TABLES contact

```
ATTRIBUTES
    c01 = contact.cdate, DEFAULT = TODAY, AUTONEXT;
    c02 = contact.empnum, AUTONEXT;
    c03 = contact.ndate, AUTONEXT;
    c04 = contact.nemp, AUTONEXT; INCLUDE clause
    c05 = contact.notes, INCLUDE = (C, I, Q, S), AUTONEXT;
    c = contact.ctype, UPSHIFT, INCLUDE = (C, I, Q, S), AUTONEXT;
```

INSTRUCTIONS

DELIMITERS " "

Figure 1-17: The Code for the f_scontact Form

ATTRIBUTES section relates the field tags to columns in tables of the database and can also specify edit-checking criteria, as well as default values. The square brackets normally appear on the form except when, as in the case of the form in Figure 1-17, the DELIMITERS clause appears in the INSTRUCTIONS section.

The ATTRIBUTES section shown in Figure 1-17 assigns the default value of TODAY (today's date) to the field represented by the tag c01. The field represented by the tag c gets shifted automatically to uppercase and allows the user to enter in the field only the values specified by the INCLUDE clause. Most of the fields use the AUTONEXT attribute to move the cursor automatically to the next field on the form when the user fills the field. As the user is entering data, the cursor moves from field to field in the order in which the tags are listed in the ATTRIBUTES section.

The INSTRUCTIONS section specifies that spaces are to be used as field delimiters when the form is displayed (the default is square brackets).

INPUT BY NAME: Moving Data from the Screen to the Program

The INPUT BY NAME statement shown in Figure 1-18 is the INFORMIX-4GL code that works with the form shown in Figures 1-10 and 1-17. This statement accepts the data the user enters on the form on the screen, prompting the user as necessary. As the user fills in each field on the form, the INPUT BY NAME statement copies the entry into an element of a record variable so the program can work with it.

The BY NAME clause of the INPUT statement means that the program will automatically figure out which form field it is to use to assign a value to a variable. The only restriction is that the field name in the ATTRIBUTES section of the form (Figure 1-17) and variable name in the program must have the same last part. In the example, the program assigns the contents of the screen field **contact.cdate** to the variable **pr_contact.cdate** because both the field name and the variable name end in **.cdate**.

The AFTER FIELD clause of the INPUT statement allows you to execute INFORMIX-4GL statements after a user has entered a value in a form field, just before the cursor moves to the next field. In Figure 1-18, the first AFTER FIELD clause enters a date that is two weeks (14 days) from the contact date as a suggested follow-up

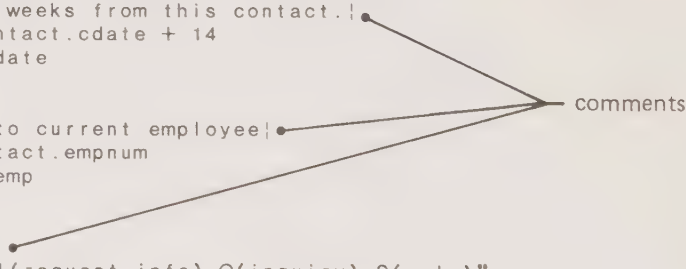
```
INPUT BY NAME pr_contact.cdate THRU pr_contact.ctype

AFTER FIELD cdate
{Set and display next contact 2 weeks from this contact.}
  LET pr_contact.ndate = pr_contact.cdate + 14
  DISPLAY BY NAME pr_contact.ndate

AFTER FIELD empnum
{Set and display next employee to current employee}
  LET pr_contact.nemp = pr_contact.empnum
  DISPLAY BY NAME pr_contact.nemp

BEFORE FIELD ctype
{Display instructions for user.}
  MESSAGE "Enter C(complaint) I(request info) Q(inquiry) S(sale)"
  ATTRIBUTE(REVERSE)

AFTER FIELD ctype
  MESSAGE ""
  {Clear message line.}
END INPUT
```



comments

Figure 1-18: The INPUT BY NAME Statement from the s_contact Program

contact date and then displays this date in the **ndate** field.

The **BEFORE FIELD** clause works in the same way as **AFTER FIELD** does, except **INFORMIX-4GL** executes the statements in a **BEFORE FIELD** clause *before* it accepts input from the user for the field. In the example, the **BEFORE FIELD** ctype clause uses a **MESSAGE** statement to display a message in reverse video that tells the user to Enter C (complaint) I (request info) Q (inquiry) S (sale).

Figure 1-18 shows comments embedded within the program. In **INFORMIX-4GL**, comments are delimited with braces ({}).

DISPLAY ARRAY: Automatic Scrolling on Display

What about the salesperson who is in the middle of entering information on a sale and forgets the correct product number? How do you get the program to display a list of products for the salesperson to choose from? The **s_contact** program is set up so that if the user enters 0 or nothing as a product code, it will open a window and display a list of product descriptions allowing the user to select one.

When the user does not enter a value in an empty field and just presses **RETURN**, **INFORMIX-4GL** assigns a null value to the variable corresponding to the field. The **AFTER FIELD** pcode clause (Figure 1-19) checks to see if the **pcode** element of the **pr_sale** record is null or equal to 0 (that is, it checks to see if the user just pressed **RETURN** without entering a value or entered 0). If the user just

```
AFTER FIELD pcode
{list products if user entered 0 or just pressed RETURN}
MESSAGE ""                                {Clear message line.      }
IF (pr_sale.pcode IS NULL                 {See if user just pressed  }
OR pr_sale.pcode = "0") THEN             {RETURN or entered 0.    }
CALL SET_COUNT(pcount - 1)              {Set up for DISPLAY ARRAY.}
OPEN WINDOW w_pdesc AT 9,17 WITH FORM "f_pdesc" ATTRIBUTE(BORDER)
MESSAGE "Use ARROW keys to move highlight, ESCAPE to make selection."
ATTRIBUTE(REVERSE)                       {Display message in reverse video.}
DISPLAY ARRAY pa_prod TO sr_con2.*       {Display array on form.  }
CLOSE WINDOW w_pdesc
LET cnt = ARR_CURR()                     {Determine which item user picked.}
LET pr_sale.pcode = pa_pcode[cnt]        {record element=array element}
DISPLAY BY NAME pr_sale.pcode            {Show product code on form.}
END IF
```

Figure 1-19: The **DISPLAY ARRAY** Statement from the **s_contact** Program

pressed **RETURN** or entered 0, the **AFTER FIELD** clause executes statements that open a window and display a list for the user to choose from.

The heart of the routine that assists the user by displaying the list of product descriptions is the

```
DISPLAY ARRAY pa__prod TO sr__con2.*
```

statement (refer back to Figure 1-19). This statement, which is embedded within the **AFTER FIELD** clause of the **INPUT BY NAME** statement, displays the contents of the program array of records (**pa__prod**) to the array of screen records (**sr__con2**).

The **DISPLAY ARRAY** statement allows the user to press the **ARROW** and **RETURN** keys to move a highlight on a list and, if necessary, automatically scrolls the list on the screen. Using this statement, you can display a list of records that will not all fit in an area of the screen at once, enabling the user to browse through a list of possible choices.

Once the user has moved the highlight so it is on top of the desired selection and has pressed the **ESCAPE** key to exit from the **DISPLAY ARRAY** statement, you can use **ARR_CURR** to determine which product the user picked from the list. The **ARR_CURR** function (this is a standard **INFORMIX-4GL** function and not a user-defined function) will return a value that is the index of the element of the array that corresponds to where the user left the highlight.

The balance of the **AFTER FIELD** clause closes the window (clearing the list of products from the screen and restoring the underlying form), assigns the product number of the selected product to the proper program variable, and displays the product number. Chapter 8 has more information on the **DISPLAY ARRAY** statement.

Manipulating the Database

Most of the database manipulation work you do will fall into the categories of creating a database and its associated tables and adding, removing, or updating data.

The **create** program not only creates the database tables that the **leads** program uses, but it also documents their structure. Normally you will never run this program directly—it is executed by the script that sets up the **leads** database and program. Refer

to Appendix C. But if you were creating your own database and program, you would need to create, compile, and execute a program such as **create** to establish your database.

The second statement in the **create** program creates the **leads** database. Following the creation of the database, the program creates the **contact** table. See Figure 1-20.

```

MAIN
{
The create program creates the database tables that the leads
program uses.  It also documents their structure.
}
CREATE DATABASE leads
CREATE TABLE contact
{
Each row in this table summarizes a contact with a prospect.
{
(
  cdate      DATE,
  empnum     SMALLINT,
  ndate      DATE,
  nemp       SMALLINT,
  notes      CHAR (50),
  ctype      CHAR (1),

  ref        INTEGER,
  cnum       SERIAL
)

```

Annotations in the diagram:

- column names** points to `cdate`, `empnum`, `ndate`, `nemp`, `notes`, `ctype`, `ref`, and `cnum`.
- column types** points to `DATE`, `SMALLINT`, `DATE`, `SMALLINT`, `CHAR (50)`, `CHAR (1)`, `INTEGER`, and `SERIAL`.
- comments** points to the curly-bracketed text: `{ Can join with sperson.empnum }`, `{ Can join with sperson.empnum }`, `{ C = complaint`, `! = request for information`, `Q = inquiry`, `S = sale }`, `{ Can join with prospect.ref }`, and `{ Can join with sale.cnum }`.

Figure 1-20: A CREATE TABLE Statement from the **create** Program

Creating a Table

To create a table, you must specify the columns that the table is to contain. Each column has a column name that must be unique to the table and a type specification that indicates what type of data you will store in the column (for example, DATE, MONEY, INTEGER, CHAR).

INPUT ARRAY: Input with Scrolling, Insert, and Delete

The INPUT ARRAY statement, together with its companion, DISPLAY ARRAY, allows you to display, choose, and manipulate the data in a database. You have already seen how DISPLAY ARRAY can help a user make a selection from a database. Figure 1-21 shows the **f_sproduct** form on the screen—this form is similar to the form

1. A PROBLEM AND ITS SOLUTION

you see when you choose **Product** from the **Administration** menu. Figure 1-22 shows the INPUT ARRAY statement from the **s__product** program. The **s__product** program is a simplified version of the **product** function—it contains no error-checking code. Although the final **product** function allows you to add, delete, and update rows in the **product** table, **s__product** only adds and deletes rows.

PRODUCT INFORMATION (no error checking version):

Use ARROWS and RETURN to move between fields and rows, ESCAPE to exit.
Press FUNCTION KEY 1 to insert a new row, FUNCTION KEY 2 to delete a row.

Code	Price	Description
10d	\$1.55	1.0" D-ring (special order)
10r	\$1.35	1.0" regular binder
15d	\$1.97	1.5" D-ring binder
20d	\$2.32	2.0" D-ring binder
25d	\$2.65	2.5" D-ring binder

Figure 1-21: The **f__sproduct** Form on the Screen

```
INPUT ARRAY pa_prod WITHOUT DEFAULTS FROM sr_product.*

AFTER INSERT
  LET idx = ARR_CURR()
  INSERT INTO product VALUES (pa_prod[idx].*)

AFTER DELETE
  DELETE FROM product WHERE pcode = pr_product.pcode

END INPUT
```

Figure 1-22: The INPUT ARRAY Statement from the **s__product** Program

Using the INPUT ARRAY statement shown in Figure 1-22, you can view, add to, and delete from the list of products in the database. Because this INPUT ARRAY statement includes a WITHOUT DEFAULTS clause, the first thing it does is to display as many elements of the array as the form will permit. Without this clause, the statement displays only default values and assumes you are entering all new values, not updating existing ones. After filling in the array, the INPUT ARRAY statement positions the cursor on the first row of the array on the screen and lets the user get to work.

While an INPUT ARRAY statement is executing (any INPUT ARRAY statement, not just the one in the example), the user can press function key F1 to open a line on the screen and insert a new row of data. Or the user can press function key F2 to remove the row of data that the cursor is on. As the ARROW keys move the cursor, the user can also type over any values to change them. Adding, deleting, and changing values on the screen only affects the array, however, and does not automatically change anything in the database. If you want to make changes to the database, you must use one of the INSERT, UPDATE, or DELETE statements discussed in the next sections.

You can use BEFORE ROW, AFTER INSERT, BEFORE DELETE, and other clauses to execute statements at any point in the execution of an INPUT ARRAY statement. There is even a clause that allows you to execute specified statements when the user presses various CONTROL and function keys—this clause is used in the full-blown **product** function discussed in Chapter 8.

Adding Data

The INSERT statement adds a row of data to a table. The INSERT statement in the **s_product** program (shown in Figure 1-23) is executed by the AFTER INSERT clause of the INPUT ARRAY statement. The AFTER INSERT clause is executed when the user has pressed function key F1, entered a row of data, and moved the cursor off the new row. The INSERT statement in the example copies data from the program array to the **product** table.

1. A PROBLEM AND ITS SOLUTION

```
INSERT INTO product VALUES (pa_prod[idx].*)
```

Figure 1-23: The INSERT Statement

The name **pa_prod[idx].*** represents one record of an array of records of variables. The statement in Figure 1-23 actually adds one row (three columns), which is composed of the three separate values represented by the record, to the **product** table.

Updating Data

The UPDATE statement changes the values of one or more columns in one or more rows of a table. Although **s_product** does not include an UPDATE statement, Figure 1-24 shows the UPDATE statement from the **product** function (refer to Appendix A for a listing of this function).

```
UPDATE product SET
  product.* = pa_prod[idx].*
WHERE pcode = pr_product.pcode
```

Figure 1-24: The UPDATE Statement

The WHERE clause in Figure 1-24 specifies that only rows (or just one row in this case) where the **pcode** column is equal to **pr_product.pcode** are to be updated. The SET clause specifies the values that the columns of the table are to take on. In Figure 1-24, **product.*** represents all the columns of the table and **pa_prod[idx].*** is a record from an array of records. Each element in the record must match a column in the table in order for this clause to work.

Deleting Data

The DELETE statement has a similar syntax to that of the UPDATE statement, except it does not specify any new values. See Figure 1-25.

```
DELETE FROM product WHERE pcode = pr_product.pcode
```

Figure 1-25: The DELETE Statement

The DELETE statement in Figure 1-25 is executed by the AFTER DELETE clause of the INPUT ARRAY statement. The AFTER DELETE clause is executed when the user presses function key F2. This statement deletes all the rows from the **product** table for which the WHERE clause is satisfied. Just as the SELECT statement will fetch all rows and the UPDATE statement will update all rows of a table if there is no WHERE clause, the DELETE statement will remove all rows from a table if it has no WHERE clause.

SUMMARY

This chapter presented a brief overview of a fourth generation language in action. It explained the relationship between the user's terminal, the database as it is stored in the computer's memory, and the **INFORMIX-4GL** program that controls and manipulates the data that travels between them. Figure 1-14 showed some of the important statements you can use to move data within an **INFORMIX-4GL** program.

The chapter presented a demonstration of **INFORMIX-4GL** by using the **leads** program—the same program this book uses throughout for examples. It touched on using menus to make the user's job easier (more in Chapter 3); the SELECT statement to retrieve information from the database (see Chapter 6); forms to assist the user and windows to expand the available screen space and provide a temporary change of context (Chapter 5); the report writer to present properly formatted information to the user (Chapter 7); the INPUT ARRAY statement to let the user add to, delete from, and update the database (Chapter 8); and various

other statements that manipulate the database (Chapter 4 and throughout the book).

Chapter 2 continues from this point by going into more detail about the systems analysis aspects of developing an application program.

After reading Chapter 1, you should have enough perspective on how 4GLs work to be comfortable reading the rest of this book.

2

DESIGNING THE APPLICATION

How do you go about actually setting up an application? This chapter explores a top-down approach that lets you design an optimal solution for a typical problem using **INFORMIX-4GL**. It starts by examining the menu as a design tool and then looks at the desired output from the application: the reports. After outlining the reports, the chapter deviates from the design process just long enough to cover some of the basic concepts of the relational database that forms the basis for many 4GLs. With these concepts fresh at hand, the chapter works on specifying the schema (the template for the data the database needs to store). Finally, this chapter covers some of the concepts of how to lay out the forms that the user will need to put data into and retrieve data from the database.

SETTING UP THE MENU

One way to go about creating a solution is to design the theoretical machine discussed in Chapter 1: Think of the computer as a do-anything machine with buttons. You press one button, and it performs a function such as printing any one of a number of reports, allowing you to enter some type of data, or displaying a form on the screen so you can update existing data. After labeling all the buttons you will need, you can start to group them. All the reports will probably fall into one group unless you decide to make subgroups such as management, accounting, and inventory reports. Depending on your application, you might group all the updating functions together, even if they deal with different kinds of data. Or you might want to group all functions that deal with a single kind of data (add, delete, and update one kind of data) together in one group.

When you actually implement the theoretical machine using **INFORMIX-4GL**, the groups will become menus and the labeled buttons will turn into menu selections. As you label and group the selections, think of what functions the user will normally tend to use at one time. You might also want to put less frequently used functions in a submenu or even a sub-submenu so the user can find more frequently used functions easily. When you examine the menu structure for the **leads** program in Figure 2-1, you can see that the **Administration** selection segregates the less frequently used functions from those that are used more frequently.

Keep on labeling and grouping until you feel the result accurately defines the product you want to build. The groups of labels should form a hierarchy, as will the selections within the menus. General divisions, or categories, of work the system can perform will appear at the top of the hierarchy. The selections at the bottom levels will be the specific tasks you can do.

The leads Menu

Figure 2-1 shows the layout of the menu for the **leads** program. This menu is derived directly from the list of theoretical buttons on page 11. The menu you see when you first call a program is

frequently called the top-level menu, but it does not have to be. Although the need for the **Administration** selection to be a menu may not be immediately obvious because it does not fill a primary need of the Associated Binder Company, it is a necessary part of the menu structure. After you create the structures to store information, you will need a way to update this information (suppose you add a salesperson or change the name of a product). The two selections in the **Administration** menu give you a way to change salesperson and product data.

TOP-LEVEL

CONTACT - Log information about a contact with a prospect.

REPORT - Run a report.

 Mailing-labels - Generate labels for a prospects mailing.

 Sales - Run report on sales by salesperson by month.

 Follow-up - Run report on follow-up calls that need to be made.

 Letters - Generate follow-up letters for contacts.

UPDATE - Update information about a contact or prospect.

 Contact - Correct information about a contact.

 Prospect - Update information about a prospect.

ADMINISTRATION - Change information about a product or salesperson.

 Product - Change information about a product.

 Salesperson - Change information about a salesperson.

 Add - Add a new salesperson.

 Update - Update information about a salesperson.

 Delete - Delete an old salesperson.

 List - Display a list of salespeople.

Figure 2-1: The Structure of the **leads** Menu

Once you have labeled and grouped the menu selections, you can implement them as an **INFORMIX-4GL** menu, complete with help lines. Although the menu by itself cannot work with the data the finished program will handle, it can be useful as a structural prototype. Chapter 3 explains how to create, compile, and run the prototype menu. Once the menu is running, see how it feels. Let some of the people who will be using it try it out and see if the labels and grouping make sense to them. Explain that this is a rough sketch of the final program. Find out if the users feel that this menu structure will fulfill all their needs and whether they think the functions are logically grouped.

PLANNING THE REPORTS

With the menu structure in place, the next step is to design the program output. All the output is grouped under the general category of reports, even though some of it may not be what you would normally think of as a report. Some examples of atypical reports are checks, mailing labels, invoices, and personalized form letters. Regular reports include sales analysis, inventory usage, and financial statement reports.

The simplest report in the **leads** program is the mailing label report. The initial sketch of the report is shown in Figure 2-2. The data this report needs is just what you would type as an address on an envelope.

The convention for naming reports in the **leads** program is that the report name begins with **r__** (to indicate it is a report).

Report: mailing labels for all prospects in ZIP code order
Name: r__label
Sample:

Dr. Mary Slade
CARLTON ADVERTISING
645 Avenue of the Americas
New York, NY 10023

Data needed:
 prospect
 first name
 last name
 title (Dr., Mr., Ms., etc.)
 company name
 address line 1
 address line 2 (omit if blank)
 address line 3 (omit if blank)
 city
 state
 zip

Figure 2-2: Sketch of the Mailing Label Report

As you design each of the reports, make a list of the data that is in each one and, if it is not raw data, where it comes from. Total monthly sales figures might, for example, come from daily sales reports or from individual orders. The totals and subtotals of figures contained within the report itself are not an issue at this point—the report writer can handle aggregate values once it has the data that makes up the totals.

The next report is more complex. It shows, by salesperson, the dollar volume of sales over a six-month period. At this stage, all you are trying to figure out is what kinds of names and numbers the report will contain. Exact titles and layout can wait for the final implementation.

The list of data needed in Figure 2-3 ignores the total column and row because you can derive that data from the other data in the report.

Report: dollar volume of sales over a six-month period by salesperson

Name: r__sales

Sample:

salesperson	month 1	month 2		Total
person 1	xx	xx		xxx
person 2	xx	xx		xxx
.				
.				
Total	xxx	xxx		xxxx

Data needed:

- list of salespeople

- monthly sales figures by salesperson derived from

- sales figures by date, price, quantity, discount, and salesperson

Figure 2-3: Sketch of the Sales by Salesperson Report

In the case of the Associated Binder Company's system, sales figures will be kept on a per-sale basis. Daily and monthly sales figures are derived from the individual sales themselves. For the purpose of this report, it will be necessary to ensure that, when a sale is recorded, the name or some identification of the

2. DESIGNING THE APPLICATION

salesperson is noted along with the other sales information. With this information, the report can derive monthly sales figures for each salesperson from the information that is recorded at the time of each sale.

Assuming the discount is stored so that 15 represents a 15 percent discount, you can calculate the value of an individual sale as:

$$\text{price} * \text{quantity} * (1 - \text{discount} / 100)$$

The report described in Figure 2-4 is a weekly report that displays raw data rather than summarized data.

Report: follow-up contacts by salesperson for the next week

Name: r_follow

Sample:

prospect contacts scheduled for Joe Fisher
between January 5 and January 11

January 5:

John Highland, XYZ COMPANY, 408-555-1212

previous contact: November 20, 1985, by Joe Fisher

source of contact: newspaper

type of contact: query

comments: away through 1/1/86 but interested in quantity

previous contact: October 4, 1985, by Cindy Smith

type of contact: request for info

January 6:

.
. .
.

Data needed:

list of salespeople

contact type and date

list of follow-up contact dates by salesperson including prospect name, company,
phone, source, and notes from past contacts including salesperson's name

Figure 2-4: Sketch of the Follow-Up Report

Report: custom form letter to prospects based on type and date of contact
Name: r_letter
Sample:

Mr. John Highland
XYZ COMPANY
123 Houston Street
New York, NY 10014

January 20, 1986

Dear Mr. Highland:

(paragraph regarding type of contact including date of contact)
(standard paragraph about company)
(paragraph regarding salesperson & how to contact)

Sincerely,

Joe Fisher, Eastern Regional Manager
Associated Binder Company

Data needed:		
	prospect	salesperson
	first name	position
	last name	first name
	title	last name
	company name	address line 1
	address line 1	address line 2 (omit if blank)
	address line 2 (omit if blank)	address line 3 (omit if blank)
	address line 3 (omit if blank)	city
	city	state
	state	zip
	zip	phone
	contact	
	type	
	date	

Figure 2-5: Sketch of the Letter Report

The report described in Figure 2-5 is a custom form letter that is sent in response to a contact.

When you lay out your reports in the manner shown in Figures 2-2 through 2-5, your needs for data become clearer. Summa-

2. DESIGNING THE APPLICATION

prospect
 first name
 last name
 title
 company name
 address line 1
 address line 2
 address line 3
 city
 state
 zip

salesperson
 position
 first name
 last name
 address line 1
 address line 2
 address line 3
 city
 state
 zip
 phone

contact
 date
 type

list of salespeople
sales figures by date, product, price, quantity, discount, and salesperson
list of follow-up contact dates by salesperson including prospect name, company, phone, source,
 and notes from past contacts including salesperson's name

Figure 2-6: Data Needed for the Reports

rizing the data that the reports need, and eliminating duplicates, you will come up with the list shown in Figure 2-6.

Only the last three entries in Figure 2-6 require a second thought. All the others are broken cleanly into groups (that will become tables) containing information on prospects, salespeople, and contacts.

Before discussing the process of designing the schema (the layout of the data that the program will store), it is necessary to understand some more about the terminology and fundamentals of the relational database management system that is the heart of many 4GLs.

UNDERSTANDING TABLES, ROWS, COLUMNS, AND JOINS

Because more and more modern relational database management systems and 4GLs are based on IBM's standard SQL (Structured Query Language), this book uses the terminology of SQL in its descriptions of data management. The following paragraphs mention equivalent terminology as they introduce SQL terms.

A *table* is a data structure that stores one group of related information. For example, the **contact** table in the **leads** program keeps track of contacts with prospective customers. A table is sometimes referred to as a *file* or *relation*.

A table has one or more *columns* (sometimes called *fields* or *attributes*). Each column contains one kind of information. In the **leads** program, the **contact** table includes the following columns:

cdate	date of this contact
empnum	employee number of salesperson making this contact
ndate	date of next contact
nemp	employee number of salesperson to make next contact
notes	notes about this contact
ctype	type of contact (sale, complaint, inquiry, information request)
ref	a number that identifies the prospect
cnum	a number that identifies this contact

A table can have zero or more *rows* (also called *records* or *tuples*). Each row contains one value for each of the columns in the table. A row in the **contact** table represents a single contact with a prospect. One row contains the date of the contact, the employee number of the salesperson making the contact, the date of the next contact, and so forth. See Figure 2-7.

	cdate	empnum	ndate	nemp	notes	ctype
Rows	06 03 1986	127	06/17/1986	127	I'm just filling in for this order	S
	12 17 1985	127	01 15 1986	127		S
	01 15 1986	125	01 29 1986	125	He wants oil proof binders	S
	⋮	⋮	⋮	⋮	⋮	⋮
	Columns					

Figure 2-7: The **contact** Table

The **contact** table must also be linked to the name, company, address, and other important information about the prospect, as well as to the name of the salesperson who made the contact. One possible way to include this information is simply to place it in the table by adding one column for each new kind of information. This method turns out to be a waste of time and space.

Suppose a salesperson talked to the same prospect ten times over the course of a year. The **contact** table would contain ten rows with the prospect's name, company name, address, and so on (see Figure 2-8)—that is, the same information would appear ten times. In the **leads** database, that information (a single row) would take up 256 bytes. Nine times that amount of space would be wasted—and that is counting only the space taken up by one prospect. Also, when information (such as the address) about a prospect changes, you would need to update the information in a lot of different places.

Taking advantage of the power of a relational database management system, you can set up a **prospect** table and put all the information about each prospect in this table one time only. Then all you need is a way of relating information about each

cdate	empnum	ndate	nemp	notes	ctype	pname	company
xx/xx/xx	xxx	xx/xx/xx	xxx	xxxxx xxxxx xx	x	John Pitzer	PITZER SHIRT SALES, INC.
xx/xx/xx	xxx	xx/xx/xx	xxx	xxxxxxxx xxx	x	John Pitzer	PITZER SHIRT SALES, INC.
xx/xx/xx	xxx	xx/xx/xx	xxx	xxxxxxxx xxx xxxx	x	John Pitzer	PITZER SHIRT SALES, INC.
xx/xx/xx	xxx	xx/xx/xx	xxx	xxxxx x xxxx	x	John Pitzer	PITZER SHIRT SALES, INC.
xx/xx/xx	xxx	xx/xx/xx	xxx	xxxxx xx xxxxx	x	John Pitzer	PITZER SHIRT SALES, INC.
xx/xx/xx	xxx	xx/xx/xx	xxx	xxxxx xxx xxxxx	x	John Pitzer	PITZER SHIRT SALES, INC.
xx/xx/xx	xxx	xx/xx/xx	xxx	xxxx xxxxxxx	x	John Pitzer	PITZER SHIRT SALES, INC.
xx/xx/xx	xxx	xx/xx/xx	xxx	xxxxx xxx xxxxx	x	John Pitzer	PITZER SHIRT SALES, INC.
xx/xx/xx	xxx	xx/xx/xx	xxx	xxxxxx xxxx x xxx	x	John Pitzer	PITZER SHIRT SALES, INC.
xx/xx/xx	xxx	xx/xx/xx	xxx	xxx xxxxx xxx xxx	x	John Pitzer	PITZER SHIRT SALES, INC.

Figure 2-8: A Poorly Designed **contact** Table

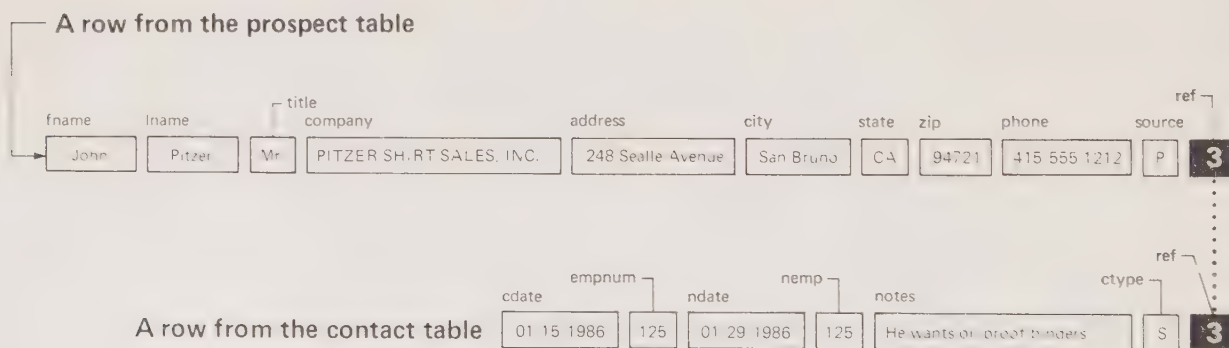
prospect to the appropriate row(s) in the **contact** table. For this scheme to function properly, each prospect must have a unique identifier so that a row in the **contact** table relates to only one row in the **prospect** table. The prospect's last name or company name would probably work as the link between these tables for a small database, but both are lengthy and cannot be guaranteed to be unique. You might have two prospects named Smith in the **prospect** table or even two companies named XYZ Company.

A clean way around the issue of length and uniqueness is to assign a unique identifier, or key, to each prospect as it is entered into the database. A serial number or other unique reference number works well in this capacity. If each row in the **contact** table has a column for the prospect reference number, the

2. DESIGNING THE APPLICATION

program can use a `SELECT` statement to retrieve the correct prospect information from the **prospect** table as needed.

Conceptually, when you use a single `SELECT` statement to retrieve information from two tables, it *joins* the tables into a single table in which each row contains columns from both tables. Usually, a `WHERE` clause specifies a relationship (called the *join condition*) between a column in one of the tables and a column in the other table so you get only the rows you need. In the case of the prospect reference number in a join between the **contact** and **prospect** tables, the `WHERE` clause specifies that the reference number in the **contact** table (**contact.ref**) must be equal to the reference number in the **prospect** table (**prospect.ref**). Thus, the `WHERE` clause ensures that you only get rows where the information about a contact with a prospect is matched with the information about the correct prospect. Refer to Figure 2-9.

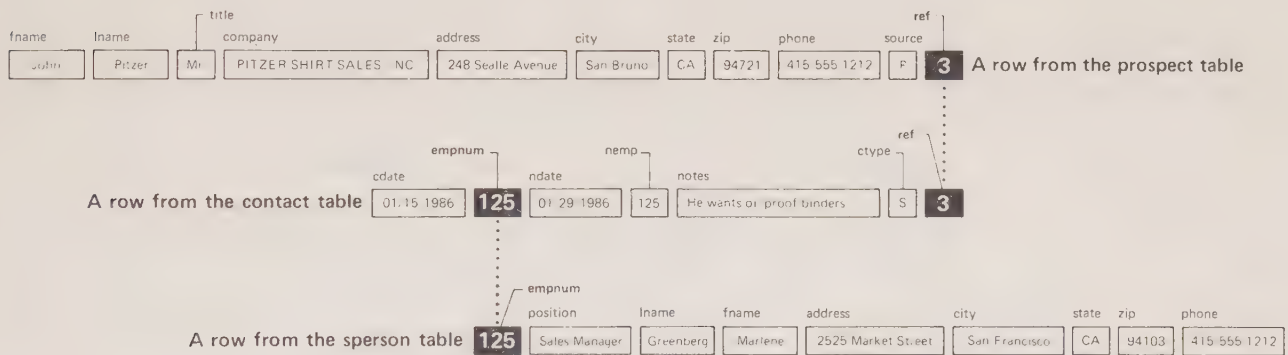


```
SELECT...WHERE prospect.ref = contact.ref
```

Figure 2-9: A Join Between the **prospect** and **contact** Tables

The same type of situation (many rows in the **contact** table needing to contain the same information) exists for the salesperson who makes the contact, and the same solution (creating another table and using a join) works. This time a unique

identifier already exists: the salesperson's employee number. If you add an employee number column to both the **contact** and salesperson (named **sperson**) tables, you can equate rows from the two tables as needed. Figure 2-10 shows the join between the **prospect**, **contact**, and **sperson** tables. With this setup, you cannot make a direct link between the **prospect** and **sperson** tables, but you can relate them through the **contact** table.



```
SELECT...WHERE prospect.ref = contact.ref
AND sperson.empnum = contact.empnum
```

Figure 2-10: A Join Between Three Tables

DESIGNING THE SCHEMA

With a basic understanding of the structure of a relational database at hand, this section continues the discussion of what to do with the last three entries in Figure 2-6. These entries are reproduced in Figure 2-11.

2. DESIGNING THE APPLICATION

list of salespeople

sales figures by date, product, price, quantity, discount, and salesperson

list of follow-up contact dates by salesperson including prospect name, company, phone, source,
and notes from past contacts including salesperson's name

Figure 2-11: Some of the Data Needed for the Reports

The first question is, How do you come up with a list of salespeople? You can get it from the **sperson** table. Each row in this table contains the name of a salesperson. If you select all first and last names (and employee numbers if necessary) from the **sperson** table, you will have a complete list of salespeople. You do not need to keep a special list of salespeople—this list will be available from the table that is being planned.

What about sources? You could make a table out of them, but the table would have only one column in it: **source**. What to do? Start by asking yourself questions about **source**. What is it? It is the source through which the prospect found out about the company. Where does it come from? It comes from the salesperson asking the prospect on the initial contact, “Where did you find out about us?” What is it associated with? It is associated with the prospect. There will not be a lot of sources, perhaps five to ten (for example, radio, newspaper, specific trade journals, word of mouth). The solution is to include the source as part of the information about the prospect. Assign a one-letter code to each possible source to save space and promote uniformity.

The issue of space is important because you are making the source a part of each row in the **prospect** table. If you have 1000 prospects and allow enough room for The New York Times in each row, you will use up 18,000 characters of information storing The New York Times over and over again. If you use N to represent the same thing, you will save 17,000 characters of storage.

The issue of uniformity is crucial. If you are running a report based on **source**, it is tedious to deal with NYT, New York Times,

The New York Times, Times, and Newspaper all meaning the same thing. In addition, there is the issue of typos. It is much easier to force everyone to use a uniform code at the time they enter the information than to have the report deal with sorting a large list of different character strings that were all intended to mean the same thing. Another way to handle the same issue, especially if you have many sources and want to keep detailed records regarding the source of each prospect, would be to set up a **source** table and assign each source a unique code. In this case you would access data in the **source** table just as you access data in the **product** or other tables.

The next item on the list in Figure 2-11 is a list of sales figures. The easiest thing to do to take care of all these items is to start a **sale** table and include the other items within it, as shown in Figure 2-12.

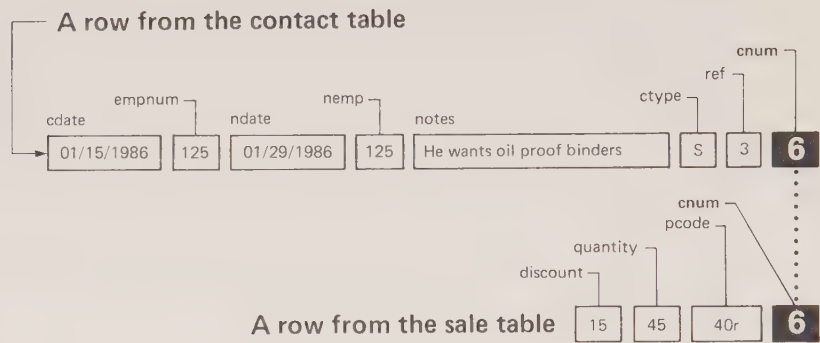
sale	
	date
	salesperson
	product
	price
	quantity
	discount

Figure 2-12: The First Pass at the **sale** Table

The problem with the **sale** table in Figure 2-12 is that it duplicates two of the **contact** table's columns: salesperson and date. There is no point in duplicating data—all you need is a way to join each row in **sale** to a row in **contact**. A unique identifier in the **contact** table is the answer. If the **sale** table has a contact number column, each row in the **sale** table can refer to its corresponding contact by including the contact number. See Figure 2-13.

The last elimination of duplicate data deals solely with the **sale** table. The **sale** table, as it stands, is shown in Figure 2-14.

2. DESIGNING THE APPLICATION



```
SELECT...WHERE sale.cnum = contact.cnum
```

Figure 2-13: A Join Between the **sale** and **contact** Tables

The product column must contain the full name of the product so a report can print it. This name represents 30 characters in each row of the **sale** table. In addition, the price of a product

sale	contact number
	product
	price
	quantity
	discount

Figure 2-14: The Second Pass at the **sale** Table

is stored repeatedly for each sale of the product. These columns waste space in the database. A solution is to create a **product** table that will contain the full name of the product (no need to skimp—it is stored only one time), the price of the product, and the obligatory unique product identifier. In many situations a unique product code already exists so you do not need to create a new identifier. The resulting **sale** and **product** tables are shown in Figure 2-15.

sale	contact number
	product code
	quantity
	discount
product	product code
	price
	description

Figure 2-15: The Final **sale** and **product** Tables

You must take into account one problem that may arise if you do not store the product price in the **sale** table. This schema does not allow for the situation in which the price of a product changes. Suppose you run a report that covers a six-month period that includes a price change. Even the sales that occurred before the price change will appear with totals that reflect the price change. One way around this problem is to store the amount of the sale in a column in the **sale** table. Or, without changing the schema you could assign a new product number to a product when its price changes. Another way to resolve this issue is to track price changes and compute the value of each sale based on the price that was in effect at the time of the sale.

The trade-offs in the price issue are storage space versus computational time versus data refinement. You can store all the raw data, including the dates price changes went into effect, discounts, and quantities. Or you can store just the value of each sale. Or you can keep track of some but not all of the raw data. With all the raw data at hand, you can always compute any number regarding past sales at any time in the future. It makes it easier to write a report a year after a series of price changes that answers the question “How did the price changes affect the dollar volume of sales?” but it means the reports will run more slowly and you will have to keep a lot more data on-line. In the **leads** program the simplest solution is used: it is assumed that a new product number will be assigned to a product when its price changes. This solution does not keep track of the dates price changes are put into effect, but it does enable you to compute the

dollar value of any sale correctly and to compare sales volumes across price changes.

Returning to the schema for the **leads** program, you still have to deal with the last entry in the list of data needed for the report (the list of follow-up contact dates by salesperson including prospect name, company, phone, and notes from past contacts including salesperson's name, shown in Figure 2-11). You can break this follow-up information down into information about the prospect (name, company, phone, and notes), the previous contact (contact date and the salesperson), and the next contact (follow-up date and salesperson). You need to add the phone number to the prospect information. To the contact information you need to add the current salesperson (using the employee number to join the salesperson information) and the future salesperson and contact date, as well as a place for notes.

The final list of data you will need is shown in Figure 2-16.

prospect		
first name	sale	
last name		contact number (Can join contact.contact number)
title		product code (Can join product.product code)
company name		quantity
address line 1		discount
address line 2		
address line 3	product	
city		product code (Can join sale.product code)
state		price
zip		description
phone		
source		
reference number (Can join contact.reference number)		
salesperson		
employee number (Can join contact.salesperson and contact.follow-up salesperson)		
position		
first name		
last name		
address line 1		
address line 2		
address line 3		
city		
state		
zip		
phone		

contact
 date
 salesperson (Can join salesperson.employee number)
 follow-up date
 follow-up salesperson (Can join salesperson.employee number)
 notes
 type
 reference number (Can join prospect.reference number)
 contact number (Can join sale.contact number)

Figure 2-16: The Final List of Data for the **leads** Program

Refer to Chapter 4 for instructions on how to convert the list in Figure 2-16 to a program that will build a database.

LAYING OUT THE FORMS

When you know what data you will need, you can begin the task of specifying the data entry and query by example forms that will appear on the user's terminal screen. The forms provide the user with an easy-to-use means of entering, manipulating, and querying the data in the database. Each form consists of fields into which **INFORMIX-4GL** or the user can enter data and text areas that identify the fields or prompt the user with suggested entries. You can design the forms to meet any objectives the project requires, from low information density to high information density. If you include only a few fields on each screen, there will be plenty of room for visual clues and suggestions, and the screen will be easy to use. The data entry process will be tedious if the data entry fields are spread over lots of screens and users have to make many screen transitions, however. On the other hand, if you put lots of fields on the screens, the high information density may make data entry more difficult.

Windows provide an excellent compromise: with windows users can input a lot of data on individual screens, without the difficulty of working with a busy, crowded screen. Because

windows pop up and are present only while they are needed, the user can enter a lot of data effectively on a single, uncrowded screen. Also, windows enable you to present the user with helpful, detailed information about the choices on a form without sacrificing valuable screen space. This is an ideal way to design a user interface that will be suitable for novices as well as experts. Novices can call up windows containing helpful information and get the assistance they need in filling in the form. Experienced users, who don't need the extra assistance, can simply fill in the form on the screen, free of the clutter of unnecessary information.

Of course, you could collect all the data on one form, but that would be inefficient. You don't want to make the user reenter the product name and price, in addition to the product code, for each sale. Similarly it should not be necessary to enter the salesperson's first name, last name, position, location, and phone number in addition to his or her employee number for each sale.

Each table will generally have one data input form associated with it, but there will be exceptions. In line with the schema that was designed in the preceding section, you will probably want to come up with separate forms for prospects, salespeople, contacts, and products. Because a sale is actually a type of contact, you will probably want to design a sale form to be opened in a window while the user is filling in the contact form. The window would open when the user enters S in the contact type field. This combination is efficient: when a salesperson is entering data about a contact with a prospect, the program has access to several pieces of data it needs in order to add a row to the **sale** table. Refer to Figure 1-10.

Another set of clues to the forms you will need comes from the menu that you set up before you designed the reports. Except for menu selections that run reports, most menu selections will require a form. In some cases, several menu selections will share a common form. For example, in the **Salesperson** sub-submenu in the **menu** program (refer to Appendix A for a listing), the **Add**, **Update**, and **Delete** selections all use the same form because they all work with the same set of salesperson information. The shared form is shown in Figure 2-17.

SALESPERSON INFORMATION

Employee number	127
Position	Eastern Sales Manager
Last name	Fuller
First name	Robert
Address 1	1047 Rich Avenue
Address 2	P.O. Box 742
Address 3	
City	San Francisco
State	CA
Zip	94104
Phone	415/555-1212

Figure 2-17: The `f__sperson` Form After It Is Filled In by the User

To start using the menu as a clue, think about the first menu selection on the **Top-Level** menu: **Contact**. The brief description of this selection is “Log information about a contact with a prospect.” (See Figure 1-1.) Already you have a hint that this form must work with two tables: **contact** and **prospect**. What is the relationship between these tables? Do you want to put information from both tables on a single form? You can have many contacts with a single prospect, but it would be nice if you did not have to enter all the prospect information for the second and subsequent contacts with a prospect. With many salespeople and many prospects, it is hard for all salespeople to keep track of whether anyone in the company has already talked to a given prospect. It would be nice if the program would verify a salesperson’s guess as to whether a prospect had been contacted before.

One solution to the prospect/contact issue is to have a function (in the **leads** program it is called **decide**) that makes an educated

2. DESIGNING THE APPLICATION

guess as to whether this is an initial contact with a prospect or not. Figure 2-18 shows the code for the **Contact** menu selection.

```
COMMAND "Contact" "Log information about a contact with a prospect."
  CALL decide(1)
  IF pr_prospect.ref IS NULL THEN CALL prospect(1) END IF
  IF eflag = 0 THEN CALL contact() END IF
  LET eflag = 0
```

Figure 2-18: The Code for the **Contact** Selection on the Menu

The code in Figure 2-18 calls the **decide** function (one of the **decide** forms is shown in Figure 2-19) when the user chooses **Contact** from the menu. If, after executing **decide**, the prospect reference number is null (does not have a value), then the **prospect** function is called so the user can enter prospect information. Figure 2-20 shows the form the **prospect** function uses. If the prospect reference number is not null, it means **decide** returned a prospect, so the program skips the **prospect** function and, if there are no errors, goes right to the **contact** function to collect information about the contact.

The **decide** function could work in many ways. The trade-offs are accuracy versus convenience versus input errors. If the user enters more information about the prospect, the program is more likely to be able to decide absolutely if the prospect is new, but the user's data entry job will be more tedious and the chance for data input errors will be greater. A misspelling in a name could cause the program to enter a single prospect as two different prospects. As a compromise the **leads** program uses three pieces of information to determine if a prospect is new or not: the prospect's last name, the company name, and the user's guess as to whether it is a new prospect. It takes this information and evaluates it. Figure 2-19 shows what the **f_decide1** form looks like on the screen.

If the prospect is a new one, the program uses the **f_prospect** form (Figure 2-20) to collect information.

Please enter the following
information about the prospect:

Last name [Pitzer]
Company name [PITZER SHIRT SALES, INC.]
Prospect type [O]

Figure 2-19: The f_decide1 Form

PROSPECT INFORMATION:

First name	John
Last name	Pitzer
Title	Mr.
Company name	PITZER SHIRT SALES, INC.
Address 1	248 Sealle Avenue
Address 2	
Address 3	
City	San Bruno
State	CA
Zip	94721
Phone number	415/555-1212
Source of lead	P

Figure 2-20: The f_prospect Form

The form has one field for each of the columns in the **prospect** table, except for the reference number. Because, in the case of this example, the reference number is generated by **INFORMIX-4GL** and is for the program's internal use only, it does not appear on the form.

All the other forms follow along the lines of this one. They each have one field that corresponds to each of the columns in the table to which the user has access.

SUMMARY

Systems analysis, the art of defining a problem and designing a solution, is perhaps the most critical part of building an application. To design a new system, list the functions you want the system to perform: everything from collecting data to running reports to correcting and updating existing data. Then group the functions and convert them into a hierarchical menu structure.

With the menu structure in place, you can design the reports that will be required. List the data that you will need to run the reports and group it so it can be stored logically and efficiently in a system of tables. Remove redundant data by creating tables you can join based on columns with common values. The result is the definition of your database: the schema.

Finally, design the forms that the user will use to enter and update data. Because the forms interact with the menu you have set up and therefore with the structure of the program, the design of the menu, forms, and program is an iterative process. This book will, of necessity, simplify the process. Do not be surprised if you find yourself going through several iterations as you are analyzing the problem at hand and designing and implementing a solution.

3

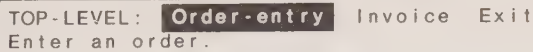
SETTING UP THE MENU

A menu is a list of selections you can choose from. Using **INFORMIX-4GL** you can build a menu that allows you to specify your selection either by pressing the key that corresponds to the first letter of your selection or by using the **SPACE** bar (or **ARROW** keys) to move a highlight from selection to selection until it is over the one you want and then pressing **RETURN**. The menu presents you with a one-line help message for each selection. As you move the highlight from one selection to the next, the message changes to correspond to the highlighted selection.

A SAMPLE MENU

This section explains how to create and compile a sample menu. Figure 3-1 shows what the sample menu looks like on the screen, and Figure 3-2 shows the **INFORMIX-4GL** code for the same menu. Below is the command line you would use to compile the menu assuming the code is stored in the **sample.4gl** file. Refer to Appendix C for more information on compiling **INFORMIX-4GL** programs.

```
c4gl -o sample sample.4gl
```



```
TOP-LEVEL:  Order-entry  Invoice  Exit
Enter an order.
```

Figure 3-1: The Sample Menu on the Screen

```
MAIN
MENU "TOP-LEVEL"
  COMMAND "Order-entry" "Enter an order."
  DISPLAY "Order entry form." AT 15,1
  COMMAND "Invoice" "Print an invoice."
  DISPLAY "Completed invoice." AT 15,1
  COMMAND "Exit" "Return to the operating system."
  EXIT MENU
END MENU
CLEAR SCREEN
END MAIN
```

Figure 3-2: The Code for the Sample Menu

The sample menu has three selections: **Order-entry**, **Invoice**, and **Exit**. The code for each of the selections starts with a **COMMAND** clause. The first string following a **COMMAND** keyword is the string that appears on the top line of the screen when the menu is displayed (**Order-entry**, **Invoice**, and **Exit** in Figure 3-2). The second string is the one-line help message that appears when the selection is highlighted—it explains what the corresponding selection does. Following these two strings are the statements

that **INFORMIX-4GL** executes when the user chooses the selection. Because the sample menu in Figure 3-2 is just a prototype, it uses simple **DISPLAY** statements in place of the statements that will appear in the final code. Another menu clause (for example, **COMMAND** or **END MENU**) terminates the list of statements.

INFORMIX-4GL passes control automatically to the current **MENU** statement after it executes the statements following a **COMMAND** clause. The third selection executes an **EXIT MENU** clause, which passes control to the statement following the **END MENU** clause for the current menu. The **CLEAR SCREEN** statement tidies up the screen before the program relinquishes control to the operating system.

Unless you use the **KEY** clause, each of the initial letters and numbers of all the selection identifiers within a single menu must be unique—the program must be able to determine which selection you want to execute from a single keystroke. The case of the letter is not significant. Refer to the *INFORMIX-4GL Reference Manual* for more information on the **KEY** clause.

Chapter 2 suggested that, after designing your system, you create a dummy menu structure as the first step of implementing the system. Figure 3-3 shows the sample menu implemented as a dummy menu structure. When you choose any one of the selections from this menu, **INFORMIX-4GL** executes the **empty** function.

```
MAIN
MENU "TOP-LEVEL"
  COMMAND "Order-entry" "Enter an order."
    CALL empty()
  COMMAND "Invoice" "Print an invoice."
    CALL empty()
  COMMAND "Exit" "Return to the operating system."
    EXIT MENU
END MENU
CLEAR SCREEN
END MAIN

FUNCTION empty()
ERROR "Function not yet implemented."
SLEEP 3
CLEAR SCREEN
END FUNCTION
```

Figure 3-3: The Code for a Dummy Menu

With a menu such as the one shown in Figure 3-3 in place, you can test the program logic (it is fairly easy to follow in this example but can become more difficult in larger, multilevel menu hierarchies). As you implement routines as **INFORMIX-4GL** functions (functions—like subroutines—are parts of the program the main routine can call), you can replace the calls to **empty** one by one with calls to working functions. The menu will work with the implemented routines and will still call **empty** for those functions that do not yet work.

As Figure 3-3 shows, a function begins with a **FUNCTION** declaration statement and ends with an **END FUNCTION** clause. You call the function with a **CALL** statement that includes the name of the function and, at a minimum, a pair of parentheses. The parentheses in the **CALL** correspond to those in the declaration; both must contain the same number and type of arguments (zero arguments are acceptable). Refer to the **sperson** function in Appendix A for an example of a function that uses parentheses to receive parameters.

The **empty** function uses an **ERROR** statement, which makes the terminal beep and displays a message on the bottom line of the screen. The program waits (using a **SLEEP** statement) for three seconds after it displays the message so the user has a chance to read the message. Then it uses a **CLEAR SCREEN** statement to get rid of the message. When control passes back to the **Top-Level** menu, **INFORMIX-4GL** redisplay the menu automatically.

SETTING UP THE **leads** MENU SYSTEM

Figure 3-4 extends the **leads** system outline that was developed in Chapter 2 by adding the necessary **Exit** selections. This particular menu system always returns control to the **Top-Level** menu when you exit from a menu at any level. You can also set up a menu system to always leave control with the current menu unless the user explicitly exits from the menu.

TOP-LEVEL

- CONTACT - Log information about a contact with a prospect.
- REPORT - Run a report.
 - Mailing labels - Generate labels for a prospects mailing.
 - Sales - Run a report on sales by salesperson by month.
 - Follow-up - Run a report on follow-up calls that need to be made.
 - Letters - Generate follow-up letters for contacts.
 - Exit - Return to the top-level menu.
- UPDATE - Update information about a contact or prospect.
 - Contact - Correct information about a contact.
 - Prospect - Update information about a prospect.
 - Exit - Return to the top-level menu.
- ADMINISTRATION - Change information about a product or salesperson.
 - Product - Change information about a product.
 - Salesperson - Change information about a salesperson.
 - Add - Add a new salesperson.
 - Update - Update information about a salesperson.
 - Delete - Delete an old salesperson.
 - List - Display a list of salespeople.
 - Exit - Return to the top-level menu.
 - Exit - Return to the top-level menu.
- EXIT - Return to the operating system.

Figure 3-4: Outline for the **leads** Menu System

The outline shows five top-level selections (**Contact**, **Report**, **Update**, **Administration**, and **Exit**). Each of the selection identifiers fulfills the requirement of starting with a different letter.

The first top-level selection, **Contact**, does not lead to another menu. When you choose **Contact**, **INFORMIX-4GL** runs the code that allows you to log information about a contact; it does not present you with another menu selection. The second selection, **Report**, leads to another menu. When you choose **Report**, **INFORMIX-4GL** displays another menu (a submenu) that lets you select which report you want to run. The **Update** selection is similar to **Report** in that it allows you to choose a function to run. The **Administration** selection allows you to choose between **Product** and the **Salesperson** menu. When you choose **Salesperson**, the system displays another menu so you can choose what (add, update, delete, or list) you want to do to the **sperson** table;

3. SETTING UP THE MENU

you can also choose to exit from the menu.

Figure 3-5 shows what the dummy menu for the **leads** program looks like. At this stage of development, the menu and its one function (**empty**) are the entire program; you can enter the code, compile it, and run it to get a sense of how control will flow once you bring up the full **leads** program.

```
DATABASE leads
GLOBALS "globals.4gl"

MAIN
MENU "TOP-LEVEL"

COMMAND "Contact" "Log information about a contact with a prospect."
    CALL empty()

COMMAND "Report" "Run a report."
    MENU "REPORT"
        COMMAND "Mailing-labels" "Generate labels for a prospects mailing."
            CALL empty()
            EXIT MENU
        COMMAND "Sales" "Run a report on sales by salesperson by month."
            CALL empty()
            EXIT MENU
        COMMAND "Follow-up" "Run a report on follow-up calls to be made."
            CALL empty()
            EXIT MENU
        COMMAND "Letters" "Generate follow-up letters for contacts."
            CALL empty()
            EXIT MENU
        COMMAND "Exit" "Return to the Top-Level menu."
            EXIT MENU
    END MENU

COMMAND "Update" "Update information about a contact or prospect."
    MENU "UPDATE"
        COMMAND "Contact" "Correct information about a contact."
            CALL empty()
            EXIT MENU
        COMMAND "Prospect" "Update information about a prospect."
            CALL empty()
            EXIT MENU
        COMMAND "Exit" "Return to the Top-Level menu."
            EXIT MENU
    END MENU

COMMAND "Administration" "Change information about a product or salesperson."
    MENU "ADMINISTRATION"
        COMMAND "Product" "Change information about a product."
            CALL empty()
            EXIT MENU
        COMMAND "Salesperson" "Change information about a salesperson."
            MENU "SALESPERSON"
```



```

COMMAND "Add" "Add a new salesperson."
  CALL empty()
  EXIT MENU
COMMAND "Update" "Update information about a salesperson."
  CALL empty()
  EXIT MENU
COMMAND "Delete" "Delete an old salesperson."
  CALL empty()
  EXIT MENU
COMMAND "List" "Display a list of salespeople."
  CALL empty()
  EXIT MENU
COMMAND "Exit" "Return to the Top-Level menu."
  EXIT MENU
END MENU
EXIT MENU
COMMAND "Exit" "Return to the Top-Level menu."
  EXIT MENU
END MENU
COMMAND "Exit" "Return to the operating system."
  EXIT MENU
END MENU
CLEAR SCREEN
END MAIN

FUNCTION empty()
  ERROR "Function not yet implemented."
  SLEEP 3
  CLEAR SCREEN
END FUNCTION

```

Figure 3-5: A Dummy Menu for **leads**

The **leads** program uses the convention of uppercase letters for menu names and lowercase letters for selection identifiers. All menu names (the name in the upper left of the screen when a menu is displayed) appear in uppercase letters so that the user can easily distinguish them from the menu selections in lowercase letters to the right.

The menu in Figure 3-5 is more complex than the sample menu. It has menus nested within menus nested within menus, and it always returns you to the **Top-Level** menu when you leave any other menu, regardless of whether you leave because you finish executing a function or because you choose an **Exit** selection.

The **Report** selection is within the **Top-Level** menu: Once you choose **Report**, **INFORMIX-4GL** executes the code between the help line (Run a report) and the next corresponding menu clause within the same menu (**COMMAND "Update"**). The code it executes

(another MENU statement) displays another menu: the **Report** menu. Before **INFORMIX-4GL** displays a new menu, such as the **Report** menu, it automatically clears whatever is on the menu (top) portion of the screen. When you choose a selection from the **Report** menu, **INFORMIX-4GL** executes the appropriate code. In the case of the dummy menu, it always calls the **empty** function.

Upward flow through the menu structure is controlled by **EXIT MENU** clauses that appear following each statement or group of statements following each selection (**COMMAND** clause). The only exception is found in the **Top-Level** menu selections. If these selections were followed by **EXIT MENU** clauses, you would always exit from the program altogether after you executed a function or chose an **Exit** selection. When you are ready to exit from the **Top-Level** menu and return to the operating system, you must choose **Exit** from the **Top-Level** menu.

If you want to change the flow of control through a submenu system so you always remain within the given submenu until you choose to exit from it, you can remove all the **EXIT MENU** clauses from the submenu. Do not, however, remove the **EXIT MENU** clauses from the **Exit** selections or you will have no way to leave the menu (except by aborting program execution).

SUMMARY

A menu system forms the basis of most **INFORMIX-4GL** programs. It allows the user to choose what he or she wants the program to do. A menu can also support various help routines. This book does not cover these help facilities—refer to the **INFORMIX-4GL User Guide** and *Reference Manual* for more information.

You can use a menu system as a prototyping tool after you design an application but before you implement it in its entirety. The prototype allows potential users to evaluate the system logic before the implementation is complete.

4

BUILDING THE SCHEMA

A *schema* is the template for the data in a database. It specifies the tables and the columns each table contains. This chapter demonstrates how to create and execute **INFORMIX-4GL** programs to set up a database. It also shows how to structure these programs so they can document the schema and remain useful even after the database is in place.

CREATING A TABLE

INFORMIX-4GL code is free format with respect to `SPACE`, `TAB`, and `RETURN` characters. The format this book uses takes advantage of this flexibility to make the code as readable as possible. This easy-to-read format allows the code that creates the database and tables also to serve as a reference document you can use as you build the rest of the system.

As an example, Figure 4-1 shows part of the database outline that was developed in Chapter 2. Figure 4-2 shows the corresponding part of the `create` program that defines and creates a database table.

```
sale
    contact number (Can join contact.contact number)
    product code (Can join product.product code)
    quantity
    discount
```

Figure 4-1: Outline of the `sale` Table

```
CREATE TABLE sale
{
Each contact that generates a sale references a row in this table
{
    (
    cnum          INTEGER,      {Can join with contact.cnum}
    pcode         CHAR (10),    {Can join with product.pcode}
    quantity      SMALLINT,
    discount      SMALLINT      {stored as an integer 1-99}
    )
}
```

Figure 4-2: The Code That Generates the `sale` Table

Figure 4-2 shows only one statement: `CREATE TABLE`. As stated previously, this book shows all `INFORMIX-4GL` keywords in upper-

case and all user-defined elements in lowercase. All elements of the program that appear between braces ({}) are comments.

The third word in a CREATE TABLE statement is the name of the table it will create. Following the table name and a comment is a pair of parentheses that holds pairs of column names and column data types, each pair separated from the next by a comma. Figure 4-3 contains a list of some of the INFORMIX-4GL data types and their characteristics.

Kind of Data	Type	Bytes	Characteristics
character	CHAR(x)	x	string of characters with length = x
integer	SMALLINT	2	whole number between -32,767 and +32,767 (inclusive)
integer	INTEGER	4	whole number between approximately -2 billion and +2 billion
floating point	SMALLFLOAT	4	floating point number
floating point	FLOAT	8	double-precision floating point number
money	MONEY	9	accurate storage of numbers with two digits to the right of the decimal—usually displayed with a \$ sign
serial	SERIAL	4	unique sequential integer automatically assigned by INFORMIX-4GL
date	DATE	4	date as in “6/5/49” or “Jun 5, 1949”

Figure 4-3: A Partial List of INFORMIX-4GL Data Types

Indexes

The judicious use of an index can increase the speed with which you can retrieve information from a database. To a lesser extent, an index can slow down the process of adding data to and updating data in a database. The question of when to create an index and when not to create one can get quite involved. Fortunately, with most 4GLs based on relational database systems, you can add and remove indexes at will. As a general rule of thumb, create an index if a table will have more than 200 rows and you will be joining it with another table. Create the index on only one of the tables that will be joined—the one that will have the most rows.

You can also create a *unique* index if you want to ensure that no two rows in a table contain the same value for a single column. With a unique index on a column, INFORMIX-4GL will not allow you to add or update a row if it duplicates the value of another row for the indexed column. Refer to the *INFORMIX-4GL User Guide* for more information on indexes.

There is a CREATE INDEX and a CREATE UNIQUE INDEX statement following the first CREATE TABLE statement in Figure 4-4. The name of the index follows the keywords CREATE INDEX (or CREATE UNIQUE INDEX), and the table and column names follow the keyword ON.

CREATING THE leads SCHEMA

Figure 4-4 shows the schema for the entire database as represented in the **create.4gl** file. To make it an INFORMIX-4GL program, all the statements must appear between MAIN and END MAIN statements. The second statement in the file creates the database that will house the tables the rest of the program creates. The **create** program will create a directory named **leads.dbs** and fill it with files that define the tables, columns, and indexes that make up the new database.

The **create** program sets up most of the columns so they will accept null values (refer to Appendix B for more information on nulls). The only columns that will not accept null values are those of type SERIAL and those that are explicitly defined as NOT NULL.

At the end of the **create** program is a GRANT statement that allows everyone to examine and change the contents of the database. See the next section of this chapter for more information on GRANT.

```

MAIN
{
The create program creates the database tables that the leads
program uses. It also documents their structure.
}
CREATE DATABASE leads

CREATE TABLE contact
{
Each row in this table summarizes a contact with a prospect.
}
(
  cdate          DATE,
  empnum         SMALLINT, {Can join with sperson.empnum}
  ndate          DATE,
  nemp          SMALLINT, {Can join with sperson.empnum}
  notes         CHAR (50),
  ctype         CHAR (1), {C = complaint
                           I = request for information
                           Q = inquiry
                           S = sale}
  ref           INTEGER, {Can join with prospect.ref}
  cnum          SERIAL   {Can join with sale.cnum}
)
CREATE INDEX i_cref ON contact(ref)
CREATE UNIQUE INDEX i_ccnum ON contact(cnum)

CREATE TABLE prospect
{
This table contains information on the prospect
(both the person and the company).
}
(
  fname         CHAR (15),
  lname         CHAR (15),
  title         CHAR (6), {letter title: Mr., Ms., Dr., etc.}
  company       CHAR (40),
  add1          CHAR (40),
  add2          CHAR (40),
  add3          CHAR (40),
  city          CHAR (40),
  state         CHAR (2),
  zip           CHAR (5),
  phone         CHAR (12),
  source        CHAR (1), {B = binder weekly
                           N = newspaper
                           O = other
                           P = binder production news
                           R = radio}
  ref           SERIAL   {This column is for the use of the program;
                           the user never sees it
                           Can join with contact.ref}
)
CREATE UNIQUE INDEX i_pref ON prospect(ref)

```

4. BUILDING THE SCHEMA

```
CREATE TABLE sale
{
Each contact that generates a sale references a row in this table
{
  (
    cnum          INTEGER,      {Can join with contact.cnum}
    pcode         CHAR (10),    {Can join with product.pcode}
    quantity      SMALLINT,
    discount      SMALLINT      {stored as an integer 1-99}
  )
}

CREATE TABLE sperson
{
Information on each salesperson is kept in this table.  The address and
phone are that of the salesperson's business location.
{
  (
    empnum        SMALLINT,      {Can join with contact.empnum or
                                {Can join with contact.nemp}
    position      CHAR (25),
    lname         CHAR (15),
    fname         CHAR (15),
    add1          CHAR (40),
    add2          CHAR (40),
    add3          CHAR (40),
    city          CHAR (40),
    state         CHAR (2),
    zip           CHAR (5),
    phone         CHAR (12)
  )
}
CREATE UNIQUE INDEX i_sempnum ON sperson(empnum)

CREATE TABLE product
{
  pcode          CHAR (10),      {Can join with sale.pcode}
  price          MONEY(6),
  descrip        CHAR (30)
}
CREATE UNIQUE INDEX i_ppcode ON product(pcode)

GRANT CONNECT TO PUBLIC
END MAIN
```

Figure 4-4: The leads Schema

All the columns shown in Figure 4-4 that are set up to relate to each other in a join are of the same type except for some that are of type SERIAL and INTEGER. Because SERIAL numbers are stored as INTEGERS, you can relate SERIAL and INTEGER columns when you join tables.

After you run the **create** program, you will get an error message if you try to run it again without dropping the **leads** database. (The term *drop* comes from the SQL DROP statements DROP

DATABASE and DROP TABLE.) **INFORMIX-4GL** will not allow you to create a database (or table for that matter) if it already exists.

When you are first building an **INFORMIX-4GL** application, you may want to make changes in your schema before you have much data in the database. One way to implement these changes is to drop the database, change the **create** program, recompile it, and run it again. You will then have a newly structured database that is accurately documented by the **create** program/schema. Once you have data in the database, you will not want to drop the database to make changes to the schema—you would lose all your data. In this situation you can use the **ALTER TABLE** statement, which changes the structure of a database without unnecessarily deleting data. Assuming the **create** program is to serve as documentation, you would then modify the **create** program so that it accurately reflects the revised structure of the database.

GRANTING PRIVILEGES

The next to last line of the **create** program (Figure 4-4) uses a **GRANT** statement to give connect privilege to everyone (public). Connect is the lowest form of database access privilege; it allows the specified person (or everyone in this case) to add, delete, and update the data in the database, but it does not allow them to modify the structure of the database in any way. A user with connect privilege cannot, for example, create or drop a table.

There are two kinds of privileges, or permissions: operating system permissions that allow specific users or the public to read from, write to, and search the operating system files that store the database data; and database privileges that grant different types of access to the database and its components. When you work with **INFORMIX-4GL**, all users can access all tables and columns of a database you create as long as the directory and file permissions at the operating system level are properly set and you have issued a **GRANT** statement, such as the **GRANT** statement in

the **create** program, so that they can *access* the database itself. If you want other users to be able to *manipulate* (change the structure of) the database, you must grant them resource privilege instead of connect privilege.

There is another type of database-related privilege, one that is associated with columns and tables (as opposed to entire databases). If you want to deny access privileges to columns or entire tables, you can use **REVOKE** statements to reverse privileges that have already been granted in order to prohibit certain users (or all other users) from accessing any particular column or table in a database.

SUMMARY

The schema is the outline of your database. You can write the program to create a database and tables in such a way that it doubles as documentation for the structure of the database. Organize the code in a logical fashion so you can see at a glance what tables contain what columns and which columns can relate to which in a join.

If other users are going to be using the database you set up, use a **GRANT** statement to make sure they have the necessary database access privileges.

5

USING FORMS

Forms make up most of a 4GL user interface. When a form appears on the screen, it presents the user with text and fields. The designer can use the text portion of a form to display a title (several forms may look alike), label the fields on the form, group related fields, separate unrelated fields, and make the form cleaner looking and easier to use.

Generally the form fields correspond to columns in a database. In a typical **INFORMIX-4GL** application, a user enters data into the fields, and the application program inserts the data into the database. Fields also allow **INFORMIX-4GL** to display data it has retrieved from the database in response to a query. Finally, fields let a user perform a query by example: The user fills some of the fields in a form, and **INFORMIX-4GL** fills in the rest from rows that contain the same values (or related values if the user entered wildcards) in the same columns as those the user filled in. Refer to Chapter 8 for more information on query by example.

In addition to being simply a place in which a user enters values, a form can perform several types of data conversion and validation. A form can ensure that the data the user enters corresponds to a certain format, it can change lowercase letters to uppercase as the user enters them in a field, it can highlight a field, and more.

Forms can also be opened in windows. A form in a window provides a convenient mechanism for displaying fields that are used only occasionally or that correspond to a distinct subgroup. They are also useful for displaying information that will assist

users in filling in the fields of a form, such as the array of product descriptions discussed in Chapter 1. Although windows are most often used with forms, they can also be used to display menus, prompts, and messages.

This chapter describes how to design forms, some of the aspects of data validation you can incorporate in your forms, and how you can open forms in windows. It also describes the INFORMIX-4GL code you need to bring the forms and windows to life.

CREATING THE FORM

The **prospect** function uses the **f__prospect** form. INFORMIX-4GL uses filename extensions to identify source code for forms and compiled forms. The name of the file that holds an uncompiled form ends in **.per**. When you compile a form, INFORMIX-4GL creates a new file with the same name but an extension of **.frm**.

```
$ form4gl -d
```

```
INFORMIX-4GL Form Builder          INFORMIX-SQL 2.1a.00
Copyright (C) Relational Database Systems, Inc., 1984-1986
Software Serial Number RDS#R000000
```

```
Enter desired name for the form: temp
Enter the name of the database used by the form: leads
```

```
You will now enter the names of the database tables which will appear on
the form. You may enter a maximum of 8 table names.
Enter a carriage return alone to terminate the sequence.
```

```
Enter a table name: prospect
Enter a table name:
```

```
The form "temp.per" will now be compiled.
```

```
    The form compilation was successful.
```

Figure 5-1: Building and Compiling a Default Form

The Default Form

The easiest form to create is a default form. A default form uses column names as field labels and does not perform any data validation. If you are using the **INFORMIX-4GL** development environment, you can use it to create a default form. Otherwise you can use the command **form4gl -d** (refer to Figure 5-1) to create a default form. The command will ask you to name the form as well as the database and table(s) the form is to be based on. Refer to Appendix C for more information on compiling forms.

Once you have created the default form, you can use it as is, or else you can edit it and then compile it again. Figure 5-2 shows the code for the default form that was created in Figure 5-1.

```

database leads
screen
{
  fname          [ f000          ]
  lname          [ f001          ]
  title          [ f002    ]
  company        [ f003          ]
  add1           [ f004          ]
  add2           [ f005          ]
  add3           [ f006          ]
  city           [ f007          ]
  state          [ a0    ]
  zip            [ f008    ]
  phone          [ f009          ]
  source         [ a    ]
  ref            [ f010          ]
}
end
tables
prospect
attributes
f000 = prospect.fname;
f001 = prospect.lname;
f002 = prospect.title;
f003 = prospect.company;
f004 = prospect.add1;
f005 = prospect.add2;
f006 = prospect.add3;
f007 = prospect.city;
a0 = prospect.state;
f008 = prospect.zip;
f009 = prospect.phone;
a = prospect.source;
f010 = prospect.ref;
end

```

Figure 5-2: The Code for the Default Form Created in Figure 5-1

Customizing a Form

Figure 5-3 shows the `f_prospect.per` file, which defines the `f_prospect` form for the `leads` database. The `f_prospect.per` file was created by using an editor to modify the default form—you can see the similarities. However, `f_prospect.per` has been modified so that all the keywords appear in uppercase, the spacing has been improved to make the file easier to read, labels have been changed to English words, and some error checking and data conversion have been incorporated.

```
DATABASE leads

SCREEN
{

PROSPECT INFORMATION:

First name      [ p01                ]
Last name       [ p02                ]
Title           [ p03                ]
Company name    [ p04                ]
Address 1       [ p05                ]
Address 2       [ p06                ]
Address 3       [ p07                ]
City            [ p08                ]
State           [ p09                ]
Zip             [ p10                ]
Phone number    [ p11                ]
Source of lead  [ p                  ]
{

TABLES prospect

ATTRIBUTES
p01 = prospect.fname, AUTONEXT;
p02 = prospect.lname, AUTONEXT;
p03 = prospect.title, AUTONEXT;
p04 = prospect.company, UPSHIFT, AUTONEXT;
p05 = prospect.add1, AUTONEXT;
p06 = prospect.add2, AUTONEXT;
p07 = prospect.add3, AUTONEXT;
p08 = prospect.city, AUTONEXT;
p09 = prospect.state, UPSHIFT, AUTONEXT;
p10 = prospect.zip, AUTONEXT;
p11 = prospect.phone, PICTURE = "###/###-###", AUTONEXT;
p    = prospect.source, UPSHIFT, INCLUDE = (B, N, O, P, R), COMMENTS =
      "B=Bind. Wkly N=Newspaper O=Other P=Bind. Prod. News R=Radio",
      AUTONEXT;
```

Figure 5-3: The Code for the `f_prospect` Form

The Parts of a Form

A form has four or five basic sections, depending on its use. The first one is the DATABASE section; it identifies the database the form works with.

The second is the SCREEN section, which specifies what the form looks like on the screen. It starts and ends with braces ({}). Any characters in this section that are not brackets ([]) and are not between brackets are taken to be literal text—**INFORMIX-4GL** displays them exactly as they appear. Brackets enclose fields. Within each pair of brackets is a field tag that, based on the following ATTRIBUTES section, defines which database column the field is associated with.

The third section declares the TABLES the form uses. The fourth section is the ATTRIBUTES section. This section associates each field tag from the SCREEN section with a column from the table(s) specified in the TABLES section. It is here you can also specify data checking and conversion. The optional fifth section (not used in the **f__prospect** form) is the INSTRUCTIONS section. It is used to define screen records and to change the default delimiters for fields.

Attributes

All the fields in the form in Figure 5-3 have the AUTONEXT attribute. This attribute causes the cursor to move from the end of one field (when it is completely filled) to the beginning of the next without the user having to press RETURN. In the **prospect** table, AUTONEXT finds most use in the fields that always contain the same number of characters (**state**, **zip**, **phone**, and **source**). The **company**, **state**, and **source** fields use the UPSHIFT attribute to ensure that all entries in these columns are in uppercase letters. The **source** field uses the INCLUDE attribute to define specifically what a user can enter in the field. If the user enters any character besides those on the list enclosed in parentheses, the program

will immediately reject it by displaying an error message and leaving the cursor on the **source** field. The form displays the string enclosed in quotation marks following the **COMMENTS** attribute while the cursor is in the field the attribute is associated with. You can use **COMMENTS** to suggest acceptable entries for a given field.

The syntax for the **ATTRIBUTES** section requires that you separate each attribute for a given field from the next attribute with a comma and that you terminate each group of attributes for a given field with a semicolon. You can group as many attributes as you like for a given field as long as they do not contradict each other (for example, **UPSHIFT** and **DOWNSHIFT**).

When the form is set up in the way you want, you need to compile it. (If you are just using the default form, you do not need to compile it—**INFORMIX-4GL** compiles the default form automatically when you build it.) You can compile an **INFORMIX-4GL** form using the **form4gl** command shown in Figure 5-4. The result will be a file with the filename extension of **.frm**—this is the file that **INFORMIX-4GL** uses. Refer to Appendix C for more information on compiling forms.

```
$ form4gl temp
```

```
INFORMIX-4GL Form Builder          INFORMIX-SQL 2.1a.00
Copyright (C) Relational Database Systems, Inc., 1984-1986
Software Serial Number RDS#R000000
```

```
The form "temp.per" will now be compiled.
```

```
    The form compilation was successful.
```

Figure 5-4: Compiling a Form

CONTROLLING THE FORM: THE INFORMIX-4GL CODE

A form works hand in hand with an INFORMIX-4GL program. The program opens the form, displays it, and displays data in the form fields, as well as accepts data from the form. The **prospect** function shown in Figure 5-5 uses the **f_prospect** form shown in Figure 5-3. The form on the screen is shown in Figure 2-20.

```
DATABASE leads
GLOBALS "globals.4gl"

FUNCTION prospect(uflag)
{
  The prospect function displays the f_prospect form.
  Depending on the calling argument, it either inserts or
  updates a row in the prospect table.

  uflag = 1 to add a new prospect,
  uflag = 2 to update an existing prospect
  {
  DEFINE      uflag          SMALLINT

  OPEN FORM f_prospect FROM "f_prospect"
  DISPLAY FORM f_prospect

  INPUT BY NAME pr_prospect.fname THRU pr_prospect.source WITHOUT DEFAULTS
  IF (uflag = 1) THEN
    LET pr_prospect.ref = 0
    INSERT INTO prospect VALUES (pr_prospect.*)
    LET pr_prospect.ref = SQLCA.SQLERRD[2]
  ELSE
    UPDATE prospect
      SET prospect.* = pr_prospect.*
      WHERE ref = pr_prospect.ref
  END IF
  CLEAR SCREEN
  END FUNCTION
```

Figure 5-5: The **prospect** Function

Variables, Declarations, Definitions, and Comments

INFORMIX-4GL does not generally allow you to work *directly* with data in the database. Instead, you declare and work with variables. The program selects data from the database and puts it

into variables and inserts data from variables into the database. The following paragraphs explain how to declare records of variables that correspond to columns in tables.

The first line in any **INFORMIX-4GL** program that works with a database must be a **DATABASE** statement that declares what database the program will be working with. The next line defines any global variables the program will use or, in the case of the **prospect** function, defines a file that contains a list of global variables. Any of the functions in **leads** that includes this **GLOBALS** statement can reference any of the variables declared in the **globals.4gl** file (shown in Figure 5-6).

```
DATABASE leads
GLOBALS
{
The globals.4gl file contains a list of global variables.
Each function that defines globals.4gl in a GLOBALS
statement can reference any of the variables declared
in this file.
}
DEFINE      pr_prospect      RECORD LIKE prospect.*,
            pr_contact      RECORD LIKE contact.*,
            pr_sale          RECORD LIKE sale.*,
            pr_sperson       RECORD LIKE sperson.*,
            pr_product       RECORD LIKE product.*,
            eflag            SMALLINT
END GLOBALS
```

Figure 5-6: The **globals.4gl** File

Record Variables

The names of record variables in the examples in this book start with **pr__** so you can tell at a glance that they are program record variables and not the names of columns, functions, tables, reports, or any other program entities. In a similar manner, most arrays of variables start with **pa__** for ease of identification. You do not have to follow these conventions when you write an **INFORMIX-4GL** program, but you may find that they make your code easier to read. Refer to Figure 1-3 for a complete list of the naming conventions used in this book.

The line in Figure 5-6 that defines **pr__prospect** uses **LIKE** to

declare this variable to be a record variable based on the **prospect** table (see Figure 5-7). This definition creates a series of variables, each beginning with **pr_prospect** and a period. Each of the variables ends with a column name—for example, **pr_prospect.fname** is the element of the record variable that corresponds to the **fname** column and **pr_prospect.source** corresponds to the **source** column. In many situations, you can refer to the entire set of variables, actually the entire record variable, with the shorthand **pr_prospect.***. When INFORMIX-4GL encounters **pr_prospect.***, it expands it into a list of all the elements of the record: **pr_prospect.fname**, **pr_prospect.lname**, **pr_prospect.title**, and so on.



Figure 5-7: The **pr_prospect** Record Variable

Using **LIKE** instead of specific types (**CHAR**, **INTEGER**, and so on) removes the responsibility for declaring the variable type from the programmer and means that a program needs only to be recompiled, not rewritten, if the structure of the data changes. The **LIKE** clause is an extremely important tool for increasing the maintainability of INFORMIX-4GL programs and should be used whenever possible.

Arrays of Records

INFORMIX-4GL also lets you define arrays of record variables. Figure 5-8 shows the part of the **DEFINE** statement from the **product** function that declares the **pa_prod** array. Figure 5-9 shows the structure of the array.

5. USING FORMS

```
DEFINE    pa_prod                ARRAY[20] OF RECORD LIKE product.*,
```

Figure 5-8: Defining an Array of Records

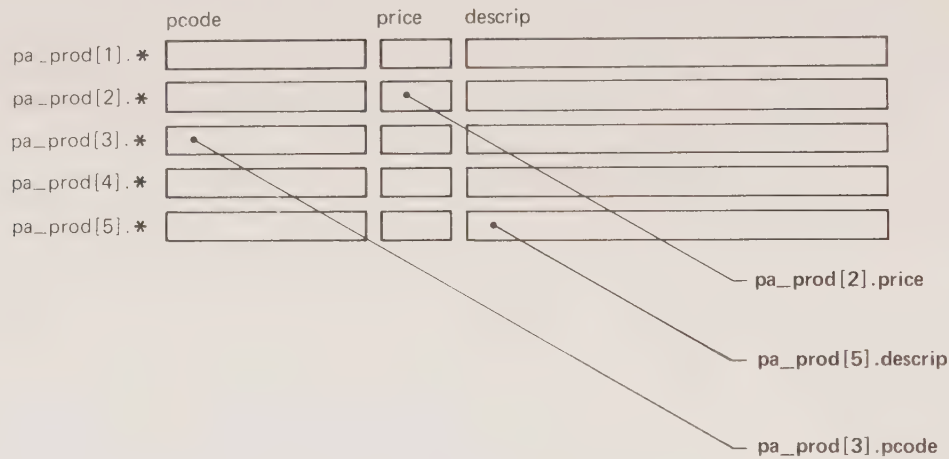


Figure 5-9: The pa__prod Array of Records

The [20] in Figure 5-8 indicates the array holds 20 records. The rest of the declaration is the standard record declaration. Each record in the array is referenced by providing an index. The variable **pa__prod[5].*** refers to the fifth record in the array. You can specify a specific element within a record by replacing the asterisk with a column name. The variable **pa__prod[5].price** refers to the value of the **price** column in the fifth record in the array. Refer to Chapter 8 for more information on the use of arrays.

The MAIN and FUNCTION Statements

The third statement of a program is usually a MAIN statement or a FUNCTION statement that declares the program to be a main routine or a function. All programs must have a main routine and can optionally include functions. You cannot execute a function

unless it is called by a main routine (or another function). Figure 5-5 shows that **prospect** is a function with one argument, **uflag**. Each call to this function must include a single value within the parentheses following the function name (refer to the **menu** program in Appendix A to see a call to the **prospect** function). The argument takes on this value so the function can use it each time it is called—in other words, the calling routine passes the value of this argument to the function it is calling.

Comments

In Figure 5-5, an eight-line comment follows the **FUNCTION** statement. You can embed comments anywhere in an **INFORMIX-4GL** program as long as each comment starts with an open brace (**{**), ends with a close brace (**}**), and does not include any other braces within the body of the comment. It is a good practice to include a comment at the top of each program module briefly describing its purpose and to include comments before complex code segments.

Local Variables

Following the comment in Figure 5-5 is the **DEFINE** statement, which defines the **uflag** variable that this function uses as an argument. All the other variables the function needs have already been defined in the **globals.4gl** file.

The **uflag** variable is *local* to the **prospect** function because it is defined between the **FUNCTION** and **END FUNCTION** statements. (It would be local to a main routine if it were defined between the **MAIN** and **END MAIN** statements.) You can access a local variable only within the routine it is local to. A global variable, on the other hand, can be accessed from any routine that contains a global declaration for the variable.

Working with the Form

The **OPEN FORM** statement associates a form name with a filename (a literal filename must be enclosed in quotation marks). It assumes the filename has an extension of **.frm**. This book uses

5. USING FORMS

the convention of beginning all form names with **f_**. The **DISPLAY FORM** statement causes **INFORMIX-4GL** to clear the screen and then display the form on the screen.

Now, with the preliminaries out of the way, the program can use the form to do its intended work. The **prospect** function assumes that some or all of the **pr_prospect.*** record has already been assigned values (it has been partially filled by the **decide** function).

The **INPUT** statement allows the user to input data into a list of program variables or elements of a record variable. The **INPUT** statement in Figure 5-10 accepts values into certain elements of record variable **pr_prospect**. The elements are designated by **pr_prospect.fname THRU pr_prospect.source**. The **THRU** stands for all the elements of the record variable between and including the named elements. (You could use a comma-separated list of each of the elements in its place.) The **pr_prospect.*** notation was not used because it would allow input in all of the **pr_prospect** fields, even the column **pr_prospect.ref**, which is of type **SERIAL**.

```
INPUT BY NAME pr_prospect.fname THRU pr_prospect.source WITHOUT DEFAULTS
```

Figure 5-10: The **INPUT BY NAME** Statement from the **prospect** Function

The **WITHOUT DEFAULTS** clause of the **INPUT** statement in Figure 5-10 causes **INFORMIX-4GL** to display on the form the values that already exist in the **pr_prospect** record and to allow the user to accept or overwrite any of the values. If you were entering all new information into a form, you would omit the **WITHOUT DEFAULTS** clause and **INFORMIX-4GL** would display default values for the appropriate form fields. In the case of the **prospect** function, even when you are inserting a new prospect into the database, some of the elements of the **pr_prospect** record already have values from the **decide** function. The **WITHOUT DEFAULTS** clause preserves these values.

The **BY NAME** clause causes **INFORMIX-4GL** to associate elements of the record variable with the appropriate form fields based on the column name and the last part of the variable name.

Getting the Data into the Table

The **prospect** function allows the user to modify the **prospect** table. It is called when the user chooses **Contact** from the **Top-Level** menu and needs to enter information about a new prospect—in this case the function uses an **INSERT** statement to add a new row to the table. It is also called when the user chooses **Prospect** from the **Update** menu—here the function uses an **UPDATE** statement to change the values in one or more columns in a row that already exists in the table.

Following execution of the **INPUT** statement, **pr_prospect.*** contains all the values entered by the user. Next, the function checks to see if **uflag** is equal to 1, indicating that the user has entered data on a new prospect. The value of **uflag** is passed by the calling program to **prospect** as an argument. If it is a new prospect, **prospect** must add a new row to the table so it executes the statements following **THEN**. If the prospect is not new, it is assumed to be a prospect whose information needs to be updated, and the statements following the **ELSE** are executed. Refer back to Figure 5-5.

In order to take advantage of **INFORMIX-4GL**'s capability to assign sequential serial numbers to a column of type **SERIAL** automatically, the corresponding value in the **INSERT** statement must be 0. The first statement following **THEN** is a **LET** that assigns 0 to **pr_prospect.ref**. The next statement is the standard SQL **INSERT** statement that inserts a new row into a table. It inserts the values stored in **pr_prospect.*** into the **prospect** table. (You can now use the asterisk to represent all the elements of the record because the statement is inserting values for all columns of the table.) The elements that you insert into a table must be in the same order as they occur in the table. Because the **pr_prospect** record was defined to be like the **prospect** table, its elements are automatically in the proper order.

The **contact** function, which is frequently called after the **prospect** function, needs to have access to the **ref** column. But, for a new row, this column has a value of 0 in the **pr_prospect** record. The **LET** statement following the **INSERT** statement assigns the

value of the **INFORMIX-4GL** variable **SQLCA.SQLERRD[2]** to **pr__prospect.ref**. After an execution of an **INSERT** statement with a serial number, this variable has the value of the serial number assigned by **INFORMIX-4GL**.

If the table is to be updated, the **UPDATE** statement does the job. When you use an **UPDATE** statement, you must use a **WHERE** clause unless you want to update every row in the table. The **WHERE** clause can specify one or more rows for updating. The **WHERE** clause in the example specifies that **INFORMIX-4GL** is to update just the row in the table where the **ref** column has the same value as **pr__prospect.ref**. (You do not need to precede the column name with the table name in this case because there is no ambiguity as to which table the column belongs to.) This choice of rows is appropriate because it is the same row in which the user has just finished changing values. The heart of the **UPDATE** statement is the **SET** clause, which in the example sets each of the columns (**prospect.***) equal to the corresponding element of the record variable (**pr__prospect.***).

After the row has been inserted or updated, the **prospect** function clears the screen and ends with the required **END FUNCTION** clause.

OPENING THE FORM IN A WINDOW

INFORMIX-4GL lets you display forms in windows. The **contact** function uses windows to display forms containing information that will help the user fill in data about a contact with a prospect. When filling in data about a contact, if the user enters a 0 in the salesperson or next salesperson field, the **contact** function opens a window and displays a form containing a list of salespeople to choose from (see Figure 5-11). The window temporarily obscures most of the **f__contact** form. Once the user selects a salesperson by positioning the cursor and pressing **ESCAPE**, the window disappears and **INFORMIX-4GL** automatically restores the underlying **f__contact** form.

Enter 0 to display and choose from a list of values.

CONTACT INFORMATION:

Ne

Use ARROW keys to move highlight, ESCAPE to make selection.

Jerome Bell
Robert Fuller
Marlene Greenberg
Molly Kimball
Isabelle Richardson

Figure 5-11: The `w__empnum` Window on Top of the `f__contact` Form

The OPEN WINDOW Statement

Figure 5-12 shows the OPEN WINDOW statement that opens the `w__empnum` window shown in Figure 5-11. The OPEN WINDOW statement attaches the name `w__empnum` to the window. This book uses `w__` as a prefix for window identifiers.

```
OPEN WINDOW w__empnum AT 6,10 WITH FORM "f__empnum" ATTRIBUTE(BORDER)
```

Figure 5-12: The Code That Opens the `w__empnum` Window

The `AT 6,10` clause identifies the window's position: `w__empnum` has its upper left corner at row 6, column 10 of the full screen.

The `WITH FORM "f__empnum"` clause causes INFORMIX-4GL to open and display the `f__empnum` form in the window. If you use the `WITH FORM` clause, the window automatically handles form management and you cannot use the `OPEN FORM`, `DISPLAY FORM`, or `CLOSE FORM` statements in reference to the form.

The WITH FORM clause causes INFORMIX-4GL to size the window to fit the form automatically. In determining the size of the window, INFORMIX-4GL includes two lines at the top of the form and one line below it. These lines are reserved for text produced by the PROMPT and MESSAGE statements and the COMMENTS attribute, respectively. The left edge of the window is placed at the left edge of the form, and the right edge is placed adjacent to the last character on the form. On most forms, the last character is an end-of-field delimiter. If you want to make the window wider, you can put spaces in the form beyond the rightmost delimiter to extend the width of the form. The `f_empnum` form, shown in Figure 5-13, has spaces inserted to the right of the first field to make the window wide enough for `msg1`, the message that is presented at the top of the window (see Figure 5-11). The spaces are followed by a period to show you where they end. (You don't need to use the period in your forms.) If these spaces were not added to the form, the window would not be wide enough for `msg1` and the message would be truncated at the right edge of the window.

```
DATABASE FORMONLY
SCREEN
{
    [ c06                                     ]
    [ c06                                     ]
    [ c06                                     ]
    [ c06                                     ]
    [ c06                                     ]
}

ATTRIBUTES
    c06    = FORMONLY.flname;

INSTRUCTIONS
    DELIMITERS    "    "
    SCREEN RECORD sr_con1[5] (FORMONLY.flname);
```

Figure 5-13: The Code for the `f_empnum` Form

If you want a window sized larger than the window INFORMIX-4GL automatically creates, or if you want to use several forms in a window, you can include a WITH *x* ROWS, *y* COLUMNS

clause in the OPEN WINDOW statement. This clause specifies a window's dimensions, and it replaces the WITH FORM clause. If you choose to specify the dimensions in this way, you will have to include OPEN FORM and DISPLAY FORM statements in order to use a form. Although explicit dimensioning is optional when you are presenting a form in a window, it is required when you use a MENU, DISPLAY, PROMPT, or MESSAGE statement to display information in a window without a form.

The `w_sale` window, shown in Figure 5-14, opens with explicitly defined width and height (see Figure 5-15). Because you cannot use the WITH FORM clause when you explicitly define dimensions, the code includes OPEN FORM and DISPLAY FORM statements. The specified width is large enough to accommodate the form as well as the displayed message. Because the dimensions of the `w_sale` window are determined by the OPEN WINDOW statement rather than by the content of the form, the `f_sale` form does not have spaces to the right of any field, unlike the `f_empnum` form.

CONTACT INFORMATION:

Ne

SALES INFORMATION:

Product code	20d
Quantity sold	15
Discount	10

Figure 5-14: The `w_sale` Window on Top of the `f_contact` Form

5. USING FORMS

```
OPEN WINDOW w_sale AT 6,10
  WITH 9 ROWS, 55 COLUMNS
  ATTRIBUTE(BORDER)
OPEN FORM f_sale FROM "f_sale"
DISPLAY FORM f_sale
```

Figure 5-15: The Code That Opens the `w__sale` Window

The `ATTRIBUTE` clause in the `OPEN WINDOW` statement includes a border around the window to make it visually distinct from the background. The border appears in the rows above and below the window as well as in columns to the left and right. Because the `w__sale` window uses rows 6 through 14 and columns 10 through 64, the border is drawn in rows 5 and 15 and columns 9 and 65. (On some terminals, the border appears in columns 8 and 66.)

You can also use an `ATTRIBUTE` clause to display the text and fields in a window in color. The `OPEN WINDOW` statement in Figure 5-16 opens a window in red, but it could as easily have used white, yellow, magenta, cyan, green, blue, or black. The `MESSAGE`, `ERROR`, `DISPLAY`, `DISPLAY FORM`, and `DISPLAY ARRAY` statements can also use color `ATTRIBUTE` clauses. Because colors catch the user's attention, you can use them to distinguish different kinds of information or to set off particularly significant messages.

```
OPEN WINDOW w_pdesc AT 9,17 WITH FORM "f_pdesc" ATTRIBUTE(BORDER,RED)
```

Figure 5-16: An `ATTRIBUTE` Clause Specifying Color

The `CLOSE WINDOW` statement in Figure 5-17 removes the `w__sale` window from the screen and restores the underlying screen and the previous application context.

```
CLOSE WINDOW w_sale
```

Figure 5-17: The Code That Closes the `w__sale` Window

Multiple-Window Programs

With INFORMIX-4GL you can open several windows at once, placing them either overlapping one another or in adjacent positions, effectively tiling the screen. Tiling is useful if the user needs to look at information in one window while working with data in another. If the user's attention will be focused only on the information displayed in a single window, overlapping windows are preferable. The **contact** function displays two overlapping windows, **w_sale** and **w_pdesc**. If the user enters a 0 in the product code field in the **w_sale** window, the program opens the **w_pdesc** window on top of the **w_sale** window, obscuring much of it (see Figure 5-18). Once the user selects a product description, the **w_pdesc** window closes and the underlying **w_sale** window is restored.

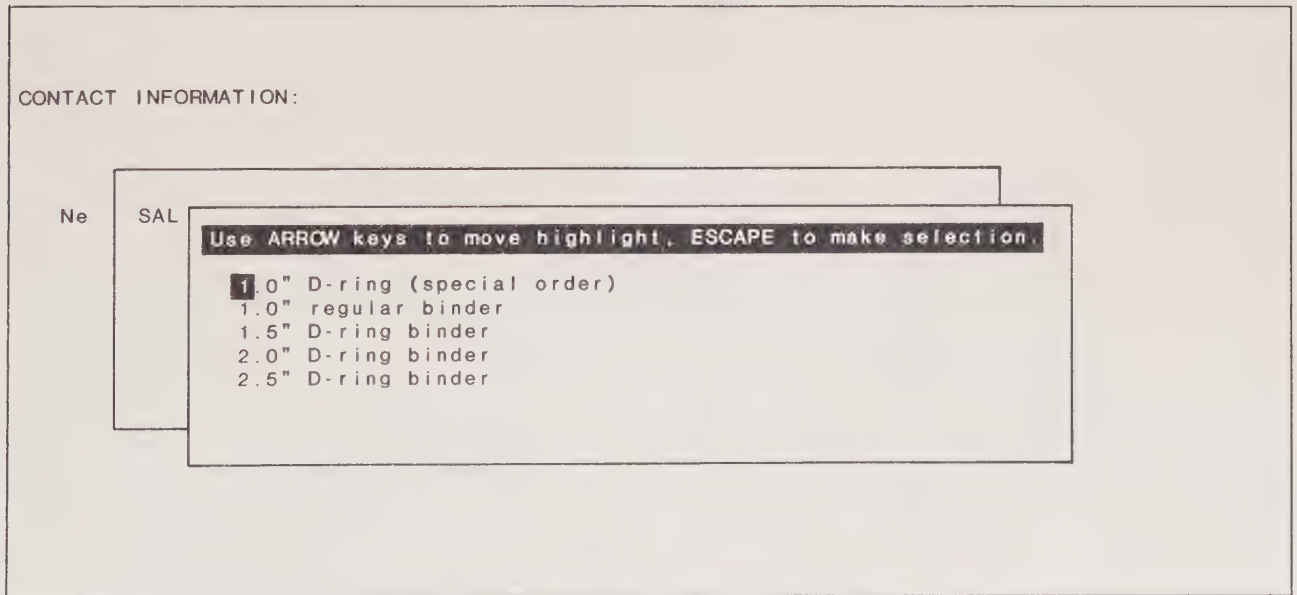


Figure 5-18: The **w_pdesc** Window Opened on Top of the **w_sale** Window

The Current Window

To facilitate working with multiple windows, **INFORMIX-4GL** maintains a list of the currently open windows. As you open each new window, it is added to the front of the list. When a window is the first one in the list, it is the *current window*: it is fully visible and the user interacts with it exclusively. As each window is closed, it is removed from the list. Once all the windows in front of a given window have been closed, that window is at the front of the list and it becomes the current window again. For example, once the `w_sale` window has been opened, it is the current window until the `w_pdsc` window is opened. Once the `w_pdsc` window is closed, the `w_sale` window becomes the current window again.

A window can also be made the current window with the `CURRENT WINDOW` statement. The `CURRENT WINDOW` statement moves a window to the front of the list without affecting the ordering of the other open windows.

A window can be closed whether it is the current window or not. It can also be cleared with the `CLEAR WINDOW` statement. The `CLEAR WINDOW` statement is typically used when you have closed a form in a window and want to clear the window to display other information.

SUMMARY

Most user interaction with a 4GL application takes place through forms. This chapter discussed how to create default forms quickly and how to customize them. It went into detail about the **INFORMIX-4GL** code that controls the forms—main routines and functions—and the use of different types of variables. It explained how to use the form within a program and how to take the data the user enters on the form and store it in the database. The chapter also described how to open forms in windows. It explained how you can let **INFORMIX-4GL** automatically size a window to fit a form and how you can predefine a window's dimensions. It described the details of the `OPEN WINDOW` and `CLOSE WINDOW` statements, as well as the concept of a current window and the management of multiple windows.

6

SQL AND SELECT

In the early 1970s, E. F. Codd developed SQL (Structured Query Language) at IBM. Although by name SQL is a query language, it is in fact an entire database manipulation language that is defined by the American National Standards Institute (ANSI). The SQL portion of **INFORMIX-4GL**, the language this book uses for examples, conforms to the ANSI standard. See Figure 6-1.

ALTER INDEX*	DROP SYNONYM*	START DATABASE*
ALTER TABLE*	DROP TABLE*	UNION
BEGIN WORK*	DROP VIEW*	UNLOCK TABLE*
CLOSE DATABASE*	EXECUTE*	UPDATE
COMMIT WORK	GRANT	UPDATE STATISTICS*
CREATE DATABASE*	INSERT	WHENEVER
CREATE INDEX*	LOCK TABLE*	
CREATE SYNONYM*	PREPARE*	Cursor Manipulation:
CREATE TABLE	RENAME COLUMN*	CLOSE
CREATE VIEW	RENAME TABLE*	DECLARE
DATABASE*	REVOKE*	FETCH
DELETE	ROLLBACK WORK	FLUSH*
DROP DATABASE*	ROLLFORWARD DATABASE*	OPEN
DROP INDEX*	SELECT	PUT*

*Informix Software, Inc., additions to ANSI standard SQL

Figure 6-1: INFORMIX-4GL and ANSI Standard SQL Statements

By far the most interesting and potentially complex SQL statement is **SELECT**. The **SELECT** statement is the SQL (and **INFORMIX-4GL**) statement that retrieves specific information from a database.

SIMPLE SELECT STATEMENTS

Figure 6-2 shows a simple INFORMIX-4GL program that selects a product description based on a product code. The first two lines in the program declare the database the program will work with and put INFORMIX-4GL on notice that this is the main routine and not a function or a report.

Program:

```
DATABASE leads
MAIN
DEFINE      p_desc                CHAR(40)

SELECT      descrip               {column}
  INTO      p_desc                {variable}
  FROM      product               {table}
  WHERE     pcode = "20d"         {criterion}

DISPLAY p_desc
END MAIN
```

Output:

```
2.0" D-ring binder
```

Figure 6-2: A Simple SELECT Statement

The DEFINE statement in Figure 6-2 declares a program variable named **p_desc**. The SELECT statement itself occupies four lines for readability. You can use any format you like as long as there are SPACES, TABS, or RETURNS between keywords, column names, and other program entities. The comments (enclosed in braces) point out the different parts of the statement.

The first line of the SELECT in Figure 6-2 specifies the name of the column that the SELECT statement will return from the row or rows it fetches. In the example, the column name is **descrip**.

The second line (INTO) specifies where (the name of the program variable) SELECT will put the value or values from the rows it fetches. In the example, SELECT puts the value of the **descrip** column in the program variable **p_desc**.

The third line (FROM) gives the name of the table in which the column appears (**product**).

The fourth line (WHERE) gives the criterion that determines

which row or rows the SELECT fetches. In the example, the WHERE clause causes SELECT to fetch the row in which the column named **pcode** contains the value 20d. The value has quotation marks around it because it is a character string.

The lines following the SELECT display the fetched value and end the program.

In Figure 6-2, you know only one row in the **product** table has the product code that the WHERE clause specifies (20d). There can be only one row with a specific code because the schema specifies a unique index on that column. This type of SELECT is called a singleton SELECT because it can, at most, return a single row. This program would have failed if the SELECT had returned more than one row because INFORMIX-4GL would not have had any place to put more than one value.

The output from the program is shown at the bottom of the figure.

Using a Cursor

Figure 6-3 shows a SELECT that is not limited to returning one row. It is the same as the SELECT in Figure 6-2, except there is no WHERE clause. Without the WHERE clause, there is no criterion that limits which rows the SELECT fetches, so it returns a value from each of the rows in the **product** table.

Program:

```
DATABASE leads
MAIN
DEFINE    p_desc                CHAR(40)

DECLARE c_curs CURSOR FOR
  SELECT    descrip              {column}
    INTO    p_desc              {variable}
    FROM    product             {table}

OPEN c_curs
FETCH c_curs
WHILE (status != NOTFOUND)
  DISPLAY p_desc
  FETCH c_curs
END WHILE
CLOSE c_curs
END MAIN
```

6. SQL AND SELECT

Output:

```
1.0" D-ring (special order)
1.0" regular binder
1.5" D-ring binder
2.0" D-ring binder
2.5" regular binder
2.5" D-ring binder
3.5" D-ring binder
4.0" regular binder
```

Figure 6-3: A SELECT Statement That Fetches Several Rows

When the term *cursor* is used in conjunction with a SELECT statement, it has a meaning that is similar to its meaning when it is used to refer to the spot of light on a terminal. On a terminal, the cursor points to the place where the next character you type or the computer displays will appear. With a SELECT, the cursor points to the row in the database that the SELECT is about to fetch. Each time you open a cursor, **INFORMIX-4GL** positions the cursor so that it points to the first row that the SELECT will fetch. Appropriately, a **FETCH** statement causes SELECT to fetch the requested values from the row the cursor is pointing to and reposition the cursor so it is pointing to the next row. A **FETCH** statement sets the **status** variable to **NOTFOUND** when no rows are left to fetch.

The line in Figure 6-3 that precedes the SELECT statement declares the **INFORMIX-4GL** program entity **c_curs** to be the name of the cursor for the following SELECT statement. The **OPEN** statement positions the cursor on the first row (containing the first description), and the first **FETCH** assigns the value of the **descrip** column from the first row that the SELECT fetches to the variable **p_desc**. This first fetch “primes the pump” so that the logic of the following **WHILE** loop works correctly.

The **WHILE** loop executes the **DISPLAY** and **FETCH** statements within its body as long as the most recent **FETCH** has returned a new row. (The **!=** is the not equal operator.) When the cursor points past the last row that the SELECT can fetch, a subsequent **FETCH** sets the **status** variable to **NOTFOUND** and control passes out of the **WHILE** loop.

The FOREACH Loop

A FOREACH loop simplifies the coding when you want to loop through all the rows that a SELECT returns. The FOREACH loop in Figure 6-4 replaces the OPEN, FETCHes, and WHILE loop from the previous example by implicitly opening the cursor and then looping through each row SELECT returns until there are no rows left. When no rows remain, the FOREACH passes control to the statement following the END FOREACH clause.

Program:

```
DATABASE leads
MAIN
DEFINE    p_desc                CHAR(40)

DECLARE c_curs CURSOR FOR
  SELECT    descrip              {column}
           INTO    p_desc        {variable}
  FROM      product             {table}

FOREACH c_curs
  DISPLAY  p_desc
END FOREACH
END MAIN
```

Output:

```
1.0" D-ring (special order)
1.0" regular binder
1.5" D-ring binder
2.0" D-ring binder
2.5" regular binder
2.5" D-ring binder
3.5" D-ring binder
4.0" regular binder
```

Figure 6-4: The FOREACH Loop

Defining Variables with LIKE

Figure 6-5 uses a LIKE clause to ensure that variable types correspond to the columns they will receive values from. The variable `p__code` will be of the same type as the `pcode` column in the `product` table. As stated previously, the LIKE clause is an important tool for increasing the maintainability of a program.

6. SQL AND SELECT

The entity **product.pcode** refers to the **pcode** column in the **product** table. Because two columns in two different tables can have the same name, writing a column name in this way is sometimes the only way to identify it uniquely.

Program:

```
DATABASE leads
MAIN
DEFINE    p_code          LIKE product.pcode,
          p_desc          LIKE product.descrip,
          p_price         LIKE product.price

DECLARE c_curs CURSOR FOR
  SELECT    pcode,          {columns}
            descrip,
            price
  INTO      p_code,        {variables}
            p_desc,
            p_price
  FROM      product        {table}

FOREACH c_curs
  DISPLAY  p_code, p_desc, p_price
END FOREACH
END MAIN
```

Output:

10d	1.0" D-ring (special order)	\$1.55
10r	1.0" regular binder	\$1.35
15d	1.5" D-ring binder	\$1.97
20d	2.0" D-ring binder	\$2.32
25r	2.5" regular binder	\$2.41
25d	2.5" D-ring binder	\$2.65
35d	3.5" D-ring binder	\$4.75
40r	4.0" regular binder	\$4.98

Figure 6-5: Using the LIKE Clause

In Figure 6-5, the **SELECT** fetches values from three columns in each row of the **product** table: The value from **pcode** is assigned to **p_code**, the value from **descrip** to **p_desc**, and the value from **price** to **p_price**.

Using a Record Variable

Figure 6-6 uses record variables. This example uses less code to perform the same function as does the code in Figure 6-5. First,

the **DEFINE** statement declares the record variable **pr__product** to contain a list of variables, one for each of the columns in the **product** table. (As mentioned earlier, the **pr__** is a convention unique to this book and is not required; it stands for program record variable.) The **INFORMIX-4GL** program expands **product.*** to **product.pcode**, **product.descrip**, and **product.price**. When you use a record variable such as the one shown in Figure 6-6, you can address each element of the record as **record-name.column-name** (for example, **pr__product.price**).

Program:

```
DATABASE leads
MAIN
DEFINE    pr__product          RECORD LIKE product.*

DECLARE c_curs CURSOR FOR
  SELECT      *                  {columns}
             INTO    pr__product.* {variables}
             FROM    product      {table}

FOREACH c_curs
  DISPLAY pr__product.*
END FOREACH
END MAIN
```

Output:

```
10d          $1.551.0" D-ring (special order)
10r          $1.351.0" regular binder
15d          $1.971.5" D-ring binder
20d          $2.322.0" D-ring binder
25r          $2.412.5" regular binder
25d          $2.652.5" D-ring binder
35d          $4.753.5" D-ring binder
40r          $4.984.0" regular binder
```

Figure 6-6: Using a Record Variable

The **SELECT** statement in Figure 6-6 uses an asterisk in place of a list of columns; **INFORMIX-4GL** expands the asterisk into a list of all the columns in the table(s) listed in the **FROM** clause of the **SELECT** statement. The **INTO** clause puts the columns from each row that the **SELECT** statement fetches into the appropriate elements of the record variable, and the **DISPLAY** statement displays all the elements of the record variable. One drawback of using an asterisk in a **DISPLAY** statement is that it gives you no control

over how values are displayed. If you want a space between the **cost** and **descrip** columns in Figure 6-6, you have to specify the elements of the record individually and insert the appropriate spacing.

JOINING TABLES

Figure 6-7 begins to show the real power of **SELECT**, demonstrating its ability to join two tables and return rows based on entries in two similar columns (see page 50 for a discussion of the theory of joins). This example joins the **product** and **sale** tables and returns rows based on equality between values in the **pcode** columns in each table.

The **WHERE** clause specifies the relationship that is the basis for the rows that the **SELECT** fetches from the join of the two tables. It instructs **SELECT** to fetch the specified columns from the join of both tables where the product codes are equal to each other. In the case of this example, the result is that the proper description from the **product** table is paired with each contact number and quantity from the **sale** table.

This **WHERE** clause differs from the one in Figure 6-2 in that it equates columns from two tables. The **WHERE** in Figure 6-2 looked for equality between a constant and a column.

The **SELECT** in Figure 6-7 equates two columns that have the same name. This naming convention makes it clear to someone reading the code that the columns are intended to be equated. Columns do not have to have the same name in order for you to equate them. Conversely, it is not necessarily true that you can equate columns that have the same name. The point is that you can equate two columns if it makes sense to do so. You would not equate columns of types **DATE** and **MONEY**, but you might want to equate two columns of type **INTEGER**.

When you are selecting from more than one table, you must be sure to include the table name each time you reference a column

Program:

```

DATABASE leads
MAIN
DEFINE    pr_product      RECORD LIKE product.*,
          pr_sale         RECORD LIKE sale.*

DECLARE c_curs CURSOR FOR
  SELECT    descrip,
            cnum,
            quantity
    INTO    pr_product.descrip,
            pr_sale.cnum,
            pr_sale.quantity
  FROM      product,
            sale
  WHERE     product.pcode = sale.pcode

FOREACH c_curs
  DISPLAY  pr_sale.cnum,
            pr_sale.quantity,
            " ",
            pr_product.descrip
END FOREACH
END MAIN

```

Output:

```

22  200  1.0" D-ring (special order)
23  250  1.0" D-ring (special order)
10   10  1.0" regular binder
5    68  1.5" D-ring binder
14   15  2.0" D-ring binder
15   15  2.0" D-ring binder
16   75  2.0" D-ring binder
24   25  2.0" D-ring binder
2    500 2.5" regular binder
7    10  2.5" regular binder
1     5  2.5" D-ring binder
8   125  2.5" D-ring binder
9    75  2.5" D-ring binder
21   45  2.5" D-ring binder
3    45  3.5" D-ring binder
13   10  3.5" D-ring binder
20   50  3.5" D-ring binder
6    45  4.0" regular binder
12   25  4.0" regular binder
18   75  4.0" regular binder
19  125  4.0" regular binder

```

Figure 6-7: Specifying a Join

that has a name that is not unique among the tables you are selecting from. If a reference to a column is ambiguous,

INFORMIX-4GL will flag it as a run-time error.

Another type of ambiguity can occur when a column and variable have the same name. Although it is best to avoid this situation, you can force a reference within a SELECT statement to refer to a database column (and not to an INFORMIX-4GL variable with the same name) by preceding the column name with an ampersand (&).

Figure 6-8 demonstrates SELECT's ability to perform computations. This example is almost the same as the preceding one except it fetches an additional value that is represented by

quantity * price * (1 - discount/100)

This expression causes SELECT to compute the dollar value of the sale by figuring the discount as a fraction of the list price to be paid (the **sale** table stores the discount as a whole number representing the percent of discount) and then multiplying the quantity sold by the unit price and this fraction. The SELECT statement puts this computed value in the variable **p_amount**.

Program:

```
DATABASE leads
MAIN
DEFINE   pr_product      RECORD LIKE product.*,
         pr_sale          RECORD LIKE sale.*,
         p_amount        MONEY

DECLARE c_curs CURSOR FOR
  SELECT   descrip,
           cnum,
           quantity,
           quantity * price * (1 - discount/100)
  INTO     pr_product.descrip,
           pr_sale.cnum,
           pr_sale.quantity,
           p_amount
  FROM     product,
           sale
  WHERE    product.pcode = sale.pcode

FOREACH c_curs
  DISPLAY pr_sale.cnum,
         pr_sale.quantity,
         " ",
         pr_product.descrip,
         p_amount
END FOREACH
END MAIN
```


Output:

22	200	1.0"	D-ring (special order)	\$170.50
23	250	1.0"	D-ring (special order)	\$213.13
10	10	1.0"	regular binder	\$12.83
5	68	1.5"	D-ring binder	\$104.49
14	15	2.0"	D-ring binder	\$31.32
15	15	2.0"	D-ring binder	\$31.32
16	75	2.0"	D-ring binder	\$130.50
24	25	2.0"	D-ring binder	\$49.30
2	500	2.5"	regular binder	\$723.00
7	10	2.5"	regular binder	\$22.90
1	5	2.5"	D-ring binder	\$13.25
8	125	2.5"	D-ring binder	\$198.75
9	75	2.5"	D-ring binder	\$168.94
21	45	2.5"	D-ring binder	\$95.40
3	45	3.5"	D-ring binder	\$181.69
13	10	3.5"	D-ring binder	\$45.13
20	50	3.5"	D-ring binder	\$190.00
6	45	4.0"	regular binder	\$190.49
12	25	4.0"	regular binder	\$112.05
18	75	4.0"	regular binder	\$280.13
19	125	4.0"	regular binder	\$435.75

Figure 6-8: Computations Within a SELECT Statement

The SELECT in Figure 6-9 uses three tables and specifies a relationship (equality in this case) between two of them. The SELECT returns the number of rows that result from the join of the two tables where an equality is specified, multiplied by the number of rows in the third table where no relationship is specified. This is not a useful SELECT statement because it arbitrarily joins rows for which no relationship is specified. It does not make logical sense.

To understand what happens, imagine the case of selecting all rows from two tables without specifying a relationship between the tables (no WHERE clause). The SELECT will start by fetching one row from table *a* and, because you did not specify a relationship, will pair it with every row from table *b*. Then it will select the second row from table *a* and again pair it with every row from table *b*. This SELECT will fetch the number of rows that is the product of the number of rows in each of the tables involved. With 100 rows in each of two tables, a SELECT that does not specify a relationship will fetch 10,000 rows. If you have three tables each containing 100 rows, a SELECT without a specified relationship between the tables will fetch 1,000,000 rows. If you execute a SELECT and it seems to be taking an inordinate amount of time, make sure you have specified the relationships that will

6. SQL AND SELECT

Program:

```
DATABASE leads
MAIN
DEFINE pr_product      RECORD LIKE product.*,
pr_sale                RECORD LIKE sale.*,
pr_contact             RECORD LIKE contact.*,
p_amount              MONEY

DECLARE c_curs CURSOR FOR
SELECT
    descrip,
    quantity * price * (1 - discount/100),
    cdate
INTO
    pr_product.descrip,
    p_amount,
    pr_contact.cdate
FROM
    product,
    sale,
    contact
WHERE
    product.pcode = sale.pcode

FOREACH c_curs
    DISPLAY pr_product.descrip,
           p_amount,
           " ",
           pr_contact.cdate
END FOREACH
END MAIN
```

three tables
but only
one relationship

Output:

1.0" D-ring (special order)	\$170.50	01/15/1986
1.0" D-ring (special order)	\$170.50	06/01/1986
1.0" D-ring (special order)	\$170.50	02/05/1986
.		
4.0" regular binder	\$435.75	04/28/1986
4.0" regular binder	\$435.75	05/15/1986
4.0" regular binder	\$435.75	06/03/1986

Figure 6-9: A Poorly Constructed SELECT Statement

retrieve only the rows you want. (If the relationships are properly specified, check that the tables are properly indexed.)

The program in Figure 6-10 reduces the number of rows that the program in Figure 6-9 fetched by specifying a second relationship in the WHERE clause. All the joined rows that SELECT fetches must have the same product code in the **product** and **sale** tables *and* must also have the same contact number in both the **sale** and **contact** tables. The result is a list of sales with the date, dollar value, and item description for each one.

Program:

```

DATABASE leads
MAIN
DEFINE    pr_product      RECORD LIKE product.*,
          pr_sale         RECORD LIKE sale.*,
          pr_contact      RECORD LIKE contact.*,
          p_amount        MONEY

DECLARE c_curs CURSOR FOR
  SELECT    descrip,
            quantity * price * (1 - discount/100),
            cdate
            INTO    pr_product.descrip,
                   p_amount,
                   pr_contact.cdate
  FROM      product,
            sale,
            contact
  WHERE     product.pcode = sale.pcode
            AND sale.cnum = contact.cnum

FOREACH c_curs
  DISPLAY  pr_product.descrip,
           p_amount,
           " ",
           pr_contact.cdate
END FOREACH
END MAIN

```

three tables
with

two relationships

Output:

1.0" D-ring (special order)	\$170.50	04/28/1986
1.0" D-ring (special order)	\$213.13	05/15/1986
1.0" regular binder	\$12.83	01/10/1986
1.5" D-ring binder	\$104.49	12/17/1985
2.0" D-ring binder	\$31.32	12/06/1985
2.0" D-ring binder	\$31.32	01/15/1986
2.0" D-ring binder	\$130.50	02/20/1986
2.0" D-ring binder	\$49.30	06/03/1986
2.5" regular binder	\$723.00	06/01/1986
2.5" regular binder	\$22.90	12/02/1985
2.5" D-ring binder	\$13.25	01/15/1986
2.5" D-ring binder	\$198.75	10/01/1986
2.5" D-ring binder	\$168.94	01/29/1986
2.5" D-ring binder	\$95.40	06/25/1986
3.5" D-ring binder	\$181.69	02/05/1986
3.5" D-ring binder	\$45.13	04/15/1986
3.5" D-ring binder	\$190.00	05/15/1986
4.0" regular binder	\$190.49	01/15/1986
4.0" regular binder	\$112.05	03/17/1986
4.0" regular binder	\$280.13	03/15/1986
4.0" regular binder	\$435.75	04/18/1986

Figure 6-10: A Join Between Three Tables

ADVANCED SELECT STATEMENTS

Figure 6-11 takes things one step further than the previous example. It specifies three relationships to fetch values from four tables. The result is a list of sales that includes the salesperson's last name.

Program:

```
DATABASE leads
MAIN
DEFINE    pr_product      RECORD LIKE product.*,
          pr_sale         RECORD LIKE sale.*,
          pr_sperson      RECORD LIKE sperson.*,
          pr_contact      RECORD LIKE contact.*,
          p_amount        MONEY

DECLARE c_curs CURSOR FOR
  SELECT    descrip,
            quantity * price * (1 - discount/100),
            lname,
            cdate
    INTO    pr_product.descrip,
            p_amount,
            pr_sperson.lname,
            pr_contact.cdate
  FROM      product,
            sale,
            contact,
            sperson
  WHERE     product.pcode = sale.pcode
            AND sale.cnum = contact.cnum
            AND contact.empnum = sperson.empnum

FOREACH c_curs
  DISPLAY  pr_product.descrip,
            p_amount,
            pr_sperson.lname,
            pr_contact.cdate
END FOREACH
END MAIN
```

Output:

1.0" D-ring (special order)	\$170.50Greenberg	04/28/1986
1.0" D-ring (special order)	\$213.13Greenberg	05/15/1986
1.0" regular binder	\$12.83Fuller	01/10/1986
1.5" D-ring binder	\$104.49Fuller	12/17/1985
2.0" D-ring binder	\$31.32Bell	12/06/1985
2.0" D-ring binder	\$31.32Bell	01/15/1986
2.0" D-ring binder	\$130.50Greenberg	02/20/1986
2.0" D-ring binder	\$49.30Fuller	06/03/1986

2.5" regular binder	\$723.00	Greenberg	06/01/1986
2.5" regular binder	\$22.90	Greenberg	12/02/1985
2.5" D-ring binder	\$13.25	Greenberg	01/15/1986
2.5" D-ring binder	\$198.75	Bell	10/01/1986
2.5" D-ring binder	\$168.94	Greenberg	01/29/1986
2.5" D-ring binder	\$95.40	Bell	06/25/1986
3.5" D-ring binder	\$181.69	Fuller	02/05/1986
3.5" D-ring binder	\$45.13	Fuller	04/15/1986
3.5" D-ring binder	\$190.00	Bell	05/15/1986
4.0" regular binder	\$190.49	Greenberg	01/15/1986
4.0" regular binder	\$112.05	Fuller	03/17/1986
4.0" regular binder	\$280.13	Bell	03/15/1986
4.0" regular binder	\$435.75	Bell	04/18/1986

Figure 6-11: A Join Between Four Tables

The program in Figure 6-12 is functionally the same as that in Figure 6-11. The code that implements the solution in Figure 6-12 is shorter because of the use of a record variable. The advantage of a record is that you can logically group, and easily refer to, a group of variables that are used together.

Program:

```

DATABASE leads
MAIN
DEFINE  pr_input1      RECORD
        descrip        LIKE product.descrip,
        amount         MONEY,
        lname          LIKE sperson.lname,
        cdate          LIKE contact.cdate
      END RECORD

DECLARE c_curs CURSOR FOR
  SELECT  descrip,
          quantity * price * (1 - discount/100),
          lname,
          cdate
    INTO  pr_input1.*
  FROM    product,
          sale,
          contact,
          sperson
  WHERE   product.pcode = sale.pcode
          AND sale.cnum = contact.cnum
          AND contact.empnum = sperson.empnum

FOREACH c_curs
  DISPLAY pr_input1.*
END FOREACH
END MAIN

```

6. SQL AND SELECT

Output:

1.0" D-ring (special order)	\$170.50Greenberg	04/28/1986
1.0" D-ring (special order)	\$213.13Greenberg	05/15/1986
1.0" regular binder	\$12.83Fuller	01/10/1986
1.5" D-ring binder	\$104.49Fuller	12/17/1985
2.0" D-ring binder	\$31.32Bell	12/06/1985
2.0" D-ring binder	\$31.32Bell	01/15/1986
2.0" D-ring binder	\$130.50Greenberg	02/20/1986
2.0" D-ring binder	\$49.30Fuller	06/03/1986
2.5" regular binder	\$723.00Greenberg	06/01/1986
2.5" regular binder	\$22.90Greenberg	12/02/1985
2.5" D-ring binder	\$13.25Greenberg	01/15/1986
2.5" D-ring binder	\$198.75Bell	10/01/1986
2.5" D-ring binder	\$168.94Greenberg	01/29/1986
2.5" D-ring binder	\$95.40Bell	06/25/1986
3.5" D-ring binder	\$181.69Fuller	02/05/1986
3.5" D-ring binder	\$45.13Fuller	04/15/1986
3.5" D-ring binder	\$190.00Bell	05/15/1986
4.0" regular binder	\$190.49Greenberg	01/15/1986
4.0" regular binder	\$112.05Fuller	03/17/1986
4.0" regular binder	\$280.13Bell	03/15/1986
4.0" regular binder	\$435.75Bell	04/18/1986

Figure 6-12: Using a Record Variable

The GROUP BY Clause

The GROUP BY clause causes SELECT to condense into one row all rows containing the same value in the column named in the GROUP BY clause. When you use a GROUP BY clause, all the columns and/or expressions following the SELECT keyword must either appear in the GROUP BY clause or must be aggregate functions. The aggregate functions are COUNT, SUM, AVG, MAX, and MIN. When you use SELECT to choose an item that is in the GROUP BY clause, the item will remain constant through the group because that is the purpose of GROUP BY: to group rows with the same value in a specified column. When you use an aggregate function, SELECT will yield the result of that function applied to all the rows in the group.

Figure 6-13 demonstrates the SUM aggregate function and the GROUP BY clause.

Program:

```
DATABASE leads
MAIN
DEFINE    p_quantity      LIKE sale.quantity,
          p_code          LIKE sale.pcode

DECLARE c_curs CURSOR FOR
  SELECT    SUM (quantity),
            pcode
            INTO    p_quantity,
                  p_code
  FROM      sale
  GROUP BY  pcode

FOREACH c_curs
  DISPLAY p_quantity,
         " ",
         p_code
END FOREACH
END MAIN
```

Output:

```
250 25d
510 25r
105 35d
68 15d
270 40r
10 10r
130 20d
450 10d
```

Figure 6-13: The GROUP BY Clause

The SELECT in Figure 6-13 groups all the rows in the **sale** table by product code. The two items following the SELECT keyword each meet one of the previously mentioned criteria. The first, SUM (quantity), is an aggregate function, and the second, pcode, appears in the GROUP BY clause. The result is a list of the number of items sold (in **p__quantity**) and the product code for each item (in **p__code**).

Figure 6-14 is an expansion of Figure 6-13. It adds the **empnum** column to the list following the SELECT keyword and to the GROUP BY clause.

As is apparent from Figure 6-14, the GROUP BY clause does not

6. SQL AND SELECT

Program:

```
DATABASE leads
MAIN
DEFINE    p_quantity      LIKE sale.quantity,
          p_code          LIKE sale.pcode,
          p_employee      LIKE contact.empnum

DECLARE c_curs CURSOR FOR
  SELECT    SUM (quantity),
            pcode,
            empnum
            INTO    p_quantity,
                    p_code,
                    p_employee
  FROM      sale,
            contact
  WHERE     sale.cnum = contact.cnum
  GROUP BY  empnum,
            pcode

FOREACH c_curs
  DISPLAY  p_quantity,
           " ",
           p_code,
           p_employee
END FOREACH
END MAIN
```

Output:

80	25d	125
510	25r	125
55	35d	127
68	15d	127
45	40r	125
170	25d	126
10	10r	127
25	40r	127
30	20d	126
75	20d	125
200	40r	126
50	35d	126
450	10d	125
25	20d	127

Figure 6-14: A GROUP BY Clause with Two Columns

put the groups in any particular order. As a matter of fact, the order in which you specify columns in the GROUP BY clause does not affect the order in which the rows are fetched. The only way to ensure a particular order is to use the ORDER BY clause, as in Figure 6-15.

The ORDER BY Clause

Program:

```
DATABASE leads
MAIN
DEFINE    p_quantity      LIKE sale.quantity,
          p_code          LIKE sale.pcode,
          p_employee      LIKE contact.empnum

DECLARE c_curs CURSOR FOR
  SELECT    SUM (quantity),
            pcode,
            empnum
    INTO    p_quantity,
            p_code,
            p_employee
  FROM      sale,
            contact
  WHERE     sale.cnum = contact.cnum
  GROUP BY empnum,
            pcode
  ORDER BY empnum

FOREACH c_curs
  DISPLAY  p_quantity,
            " ",
            p_code,
            p_employee
END FOREACH
END MAIN
```

Output:

80	25d	125
510	25r	125
45	40r	125
75	20d	125
450	10d	125
170	25d	126
30	20d	126
200	40r	126
50	35d	126
55	35d	127
68	15d	127
10	10r	127
25	40r	127
25	20d	127

Figure 6-15: The ORDER BY Clause

INFORMIX-4GL can return rows that do not appear in the **ORDER BY** clause in any order. The program in Figure 6-15 orders by **empnum**, but not by **pcode**, so the output is in employee number order. Product codes are in an unpredictable order within each employee number. If you want the product codes to come out in order within each employee number, you can specify both columns in the **ORDER BY** clause:

```
ORDER BY empnum, pcode
```

Or, if you want employees to come out in order within each product code, you can reverse the order of the columns in the **ORDER BY** clause:

```
ORDER BY pcode, empnum
```

You can use the **DESC** (for descending) keyword to make product codes come out in reverse order:

```
ORDER BY pcode DESC, empnum
```

Subqueries and Temporary Tables

Figure 6-16 makes use of a subquery and a temporary table to return information about all salespeople who have above-average sales months with a particular product. Specifically, each row the program returns shows the employee number, year and month, product description, and total value of all sales for that employee, in that month, for that product. The query returns only those rows that show sales that are above the average for all employees, over all months, over all products. (The report will not show a good month of selling a cheap product if it is overshadowed by sales of a more expensive product.) It uses two queries (two **SELECT** statements). The first query stores its results in a temporary table named **holdit**. The second selects from **holdit** and puts the results in program variables.

Program:

```
DATABASE leads
MAIN
DEFINE      pr_rep      RECORD
            empnum      LIKE contact.empnum,
```

```

        mnth          SMALLINT,
        yr            SMALLINT,
        descrip       LIKE product.descrip,
        value         MONEY
    END RECORD

SELECT      contact.empnum,
            MONTH(contact.cdate) mnth,
            YEAR(contact.cdate) yr,
            product.descrip,
            SUM(sale.quantity * product.price * (100 - sale.discount)/100) value
FROM        contact,
            product,
            sale
WHERE       contact.cnum = sale.cnum
            AND sale.pcode = product.pcode
GROUP BY 1, 2, 3, 4
INTO TEMP holdit

DECLARE c_curs CURSOR FOR
SELECT      *
INTO        pr_rep.*
FROM        holdit
WHERE       value > (
                    SELECT      AVG (value)
                    FROM        holdit
                    )

ORDER BY empnum,
        yr,
        mnth,
        descrip

FOREACH c_curs
    DISPLAY pr_rep.empnum,
            pr_rep.mnth USING "###",
            pr_rep.yr USING "/####",
            " ",
            pr_rep.descrip,
            pr_rep.value

END FOREACH
END MAIN

```

display labels

Output:

125	1/1986	2.5"	D-ring binder	\$182.19
125	1/1986	4.0"	regular binder	\$190.49
125	4/1986	1.0"	D-ring (special order)	\$170.50
125	5/1986	1.0"	D-ring (special order)	\$213.13
125	6/1986	2.5"	regular binder	\$723.00
126	3/1986	4.0"	regular binder	\$280.13
126	4/1986	4.0"	regular binder	\$435.75
126	5/1986	3.5"	D-ring binder	\$190.00
126	10/1986	2.5"	D-ring binder	\$198.75
127	2/1986	3.5"	D-ring binder	\$181.69

Figure 6-16: Using a Temporary Table and a Subquery

INFORMIX-4GL automatically creates a temporary table when it executes a SELECT statement that uses an INTO TEMP clause. The temporary table is dropped when the program finishes execution. When the results of a SELECT statement go into a temporary table, each of the columns maintains its original column name but gets a new table name: the name of the temporary table. When you use SELECT to put aggregate values into a temporary table, you must assign a column name to the column that will contain the aggregate values. You assign a column name by using a *display label* to assign a new name to a column that results from a SELECT of aggregate values. You can also use a display label to assign a new name to a column that already has a name. When you follow the name of the column in the column list with the display label, SELECT will use that label in place of the column name.

The MONTH function on the second line of the SELECT statement in Figure 6-16 uses a display label of **mnth** for the column it creates. The YEAR function uses **yr**, and the SUM aggregate uses a display label of **value**. The complete names of the resulting columns will be **holdit.mnth**, **holdit.yr**, and **holdit.value**.

Because GROUP BY cannot refer directly to functions that appear following the SELECT keyword (for example, MONTH) or aggregates (for example, SUM), the GROUP BY clause in Figure 6-16 has a different format from previous ones. The numbers 1, 2, 3, and 4 in this clause represent relative positions of items following the SELECT keyword. The 1 represents the first item (contact.empnum); 2, the second [MONTH (contact.cdate)]; 3, the third [YEAR (contact.cdate)]; and 4, the fourth item (product.descrip).

The second SELECT statement includes a subquery in its WHERE clause. The SELECT statement that is embedded in the WHERE clause returns a single value: the average value of all sales from **holdit**. The WHERE clause causes the SELECT to fetch only those rows where the value of the **value** column is greater than the value the subquery returns. The result is that only rows representing sales of above-average dollar amounts are selected.

In the `DISPLAY` statement, the `USING` clauses force the value contained in `pr_rep.mnth` to be printed right-justified in three spaces and the value in `pr_rep.yr` to be preceded by a slash.

SUMMARY

This chapter has presented an introduction to the use of the `SELECT` statement within `INFORMIX-4GL`. When you work with an SQL-based 4GL such as `INFORMIX-4GL`, the `SELECT` statement will become one of your most-used tools. You can use `SELECT` simply to extract raw data from a database, or you can use it to sort, group, and arithmetically combine the data as it is being extracted. Although you can perform many of the data manipulations that `SELECT` will perform outside the `SELECT` statement (most notably in a report), you can usually perform them with less code within a `SELECT`.

7

USING THE REPORT WRITER

As used in this book, *report writer* or *report writing language* refers to a subset of the INFORMIX-4GL language that you can use to generate report output. *Report output* can be anything from a formal, conventional report such as a quarterly report, to a check or invoice, to a report that goes to the screen and never reaches a piece of paper. A *report* is the actual INFORMIX-4GL code that generates the report output. When the meaning is not ambiguous, *report output* is sometimes shortened to *report*.

Examples in the preceding chapter generated output using DISPLAY statements embedded in FOREACH loops. You can use SELECT to extract specific information from a database and even to order and group the information. But there are limits to what SELECT and INFORMIX-4GL can do easily without the aid of the report writer. The report writer, for example, makes it easy to break reports across pages with headers, including page numbers, on the top of each page. The report writer allows you to display information row by row, together with group, page, and column totals. The report writer gives you much finer control of the format (for example, margins, spacing, indentions, and columns) of a report than is otherwise possible.

This chapter explains how to take advantage of the power of the INFORMIX-4GL report writer when you extract information from a database.

SETTING UP A REPORT

A report writer takes row-by-row data as input and displays final, tabulated, formatted information as output. In some instances it may display one page of information for each row it gets as input (refer to the following `r_label` example). At the other extreme, a report may collect all its input before generating any output; then it may generate only summary information.

INFORMIX-4GL reports can be broken into two parts: the code that is the report itself, and the code that feeds, or drives, the report. The *driver* collects the information, processes it as much as is needed, and feeds the report a row at a time. Typical construction of an **INFORMIX-4GL** driver includes a `SELECT` statement that retrieves the data from the database and an associated `FOREACH` loop that calls the report once for each row the `SELECT` fetches.

The Report Driver

The following discussion uses the `r_label` report as an example. Chapter 2 sketched this report in Figure 2-2. Figure 7-1 shows the report output.

Ms. Jocelyn McCaffrey
HEALTHWAY NATURAL FOODS
405 Fernando Avenue
Palo Alto, CA 94306

Dr. Mike Pitts
TESTING ENGINEERS, INC.
1536 Fordham Court
Palo Alto, CA 94306

Ms. Linda Roos
THE DESIGN ASSOCIATES

•
•
•

Figure 7-1: Output of the `r_label` Report

The INFORMIX-4GL code that functions as the report driver supplies the report with the data it needs to produce each label. Figure 7-2 shows the driver portion of the report, named **rd_label**. (The prefix **rd_** stands for report driver and is a convention that is unique to this book. The associated report is named **r_label**.)

```
DATABASE leads
GLOBALS "globals.4gl"

FUNCTION rd_label()
{
The rd_label report driver retrieves prospects in zip code order
from the prospect table. It then passes the information to the
r_label report.
}
DECLARE c_label CURSOR FOR
  SELECT      *
    INTO      pr_prospect.*
    FROM      prospect
    ORDER BY  zip
DISPLAY "Running report, output going to label.out" AT 15,1
START REPORT r_label
FOREACH c_label
  OUTPUT TO REPORT r_label(pr_prospect.*)
END FOREACH
FINISH REPORT r_label
DISPLAY "Report finished, output in label.out", "" AT 15,1
SLEEP 3
DISPLAY "" AT 15,1
END FUNCTION
```

Figure 7-2: The Code for **rd_label**, the Driver for the **r_label** Report

Everything in the program shown in Figure 7-2 through the **SELECT** statement should be familiar to the reader at this point. As a courtesy to the user, because it may take a while to run the report, the program displays an informative message before it starts running the report. The **AT** clause at the end of the **DISPLAY** statement positions **DISPLAY**'s output on the fifteenth row, first column of the screen.

The **START REPORT** statement initializes the report. As the **FOREACH** loop goes through each of the rows the **SELECT** fetches, it executes an **OUTPUT TO REPORT** statement for each one. This statement is a call to the report that passes one row of information to the report in the form of arguments.

For the purpose of this report, it is not necessary to select all the columns from **prospect**: the report does not use the **phone**, **source**, or **ref** columns. Using an asterisk (*) in two places in the SELECT statement, however, takes less code and makes the code easier to read and therefore easier to maintain; the additional overhead, created by selecting the additional columns, in this case is minimal.

Following the END FOREACH clause, a FINISH REPORT statement closes out the report. Then, the program displays another message to confirm where the report output is. The null string ("") at the end of the message overwrites the end of the previous message. If it were not there, the end of the previous message would remain at the end of the new message. After waiting three seconds, the final DISPLAY uses another null string to clear the line and leave the screen ready for the next menu selection.

The Report

The report is the part of the INFORMIX-4GL code that produces the report output. The first statement in the **r_label** report is the DATABASE declaration. Although this report does not interact with the database directly, it must have access to the database to interpret the LIKE clause in the DEFINE statement properly. Refer to Figure 7-3.

Following any declarations, a report starts with a REPORT statement that includes a parenthesized argument list. The list must correspond one-for-one with the list in the OUTPUT TO REPORT statement that calls the report. The variable names do not have to be the same, but the variable types, if not the same, must be directly convertible. As an example, you can pass any string enclosed in quotation marks to a variable of type CHAR, but you cannot pass a string that includes letters to a variable of type INTEGER.

As in a function or main program, the DEFINE statement comes next. Instead of using explicit typing with data types such as CHAR and DATE, **r_label** uses indirect typing by referring to a specific table with a LIKE clause. The report defines the **rr** (pro-

```
DATABASE leads
REPORT r_label(rr)
{
The r_label report prints mailing labels for prospects.
The output is sent to a file called label.out.
}

DEFINE      rr                      RECORD LIKE prospect.*

OUTPUT
  REPORT TO "label.out"
  LEFT MARGIN 0
  TOP MARGIN 0
  BOTTOM MARGIN 0
  PAGE LENGTH 8 {change to the number of lines between the
                  top of one label and the top of the next}

FORMAT
ON EVERY ROW
  PRINT rr.title CLIPPED, 1 SPACE,
        rr.fname CLIPPED, 1 SPACE,
        rr.lname
  IF rr.company IS NOT NULL THEN PRINT rr.company END IF
  IF rr.add1 IS NOT NULL THEN PRINT rr.add1 END IF
  IF rr.add2 IS NOT NULL THEN PRINT rr.add2 END IF
  IF rr.add3 IS NOT NULL THEN PRINT rr.add3 END IF
  PRINT rr.city CLIPPED, ", ", rr.state, 2 SPACES, rr.zip
  SKIP TO TOP OF PAGE
END REPORT
```

Figure 7-3: The Code for the `r_label` Report

gram report record) record to be like the **prospect** table. Thus the report variables are always of the same type as the columns the data is selected from. Using this kind of typing has the advantage that, if you change the type of a column in the database, or even if you just change the length of a CHAR column, you will not need to make any changes to the report—you will only have to recompile it (unless of course the change necessitates a change in format of the report output).

The next part of the report, the **OUTPUT** section, is optional. If you do not include it, the report will be formatted for a 66-line (11-inch) page with a left margin of five spaces and top and bottom margins of three lines each. The report will go to the screen.

The **OUTPUT** section in `r_label` sets up an appropriate format for tractor-feed labels that are eight lines long from the top of one to the top of the next one. A left margin of 0 lets the lines on the labels be as long as possible, and top and bottom margins of 0

allow the maximum number of lines on each label. The `REPORT TO` clause causes the report output to go to the file named `label.out` for printing at a later time. If you omit this clause, the report output will go to the screen. You can use the `REPORT TO PRINTER` clause to send the output directly to the printer.

The final, and most important, section of a report is the `FORMAT` section. This section contains the `PAGE HEADER`, `PAGE TRAILER`, and `ON EVERY ROW` subsections. The `r_label` report has only an `ON EVERY ROW` subsection because, for labels, there is no page header or trailer. `INFORMIX-4GL` executes the statements within this subsection one time for each row it receives from the report driver.

The first statement in the `ON EVERY ROW` subsection is `PRINT`, which prints three elements of the `rr` record separated by spaces. The `CLIPPED` clause indicates that trailing spaces are not to be printed—trailing spaces occur when the value of a `CHAR` type variable is shorter than its defined length. The first element of `rr` that `PRINT` displays, `title`, is defined to be six characters long. If it contains `Dr.`, there will be three spaces at the end of the variable and, without `CLIPPED`, `PRINT` would display the first name three spaces past the end of the title. Instead, `CLIPPED` truncates all trailing spaces and `1 SPACE` prints a single space.

Each of the next four lines in the program prints an element of the `rr` record only if that element is not null. After printing the city, state, and ZIP code, the report executes a `SKIP TO TOP OF PAGE` statement. This statement skips enough lines so that the next line the report prints will be at the top of a new page. Based on the `PAGE LENGTH` statement in the `OUTPUT` section of the report, the top of the next page (or label in this case) is eight lines past the first line of the current page (label).

ANOTHER REPORT

As a stepping stone to the more complex `r_sales` report of the `leads` program, this part of this chapter introduces a report of medium complexity.

The report is interactive, using PROMPT statements to query the user for the period the report should cover. The report produces a list of employees with their total sales for the report period and a grand total for all sales during the report period. The output from the report is shown in Figure 7-4.

Sales Summary by Salesperson
For the Period 01/01/1986 through 02/28/1986

Employee Number	Total Sales
125	503.17
126	31.32
127	194.51
Total	729.00

Figure 7-4: Sample Output from the **r_medium** Report

The report driver, shown in Figure 7-5, uses a record named **pr_ssum** to store all the values it will pass to the report. It uses a SELECT to fetch the value of each of the sales within the report period and the employee number of the salesperson who made the sale. The WHERE clause specifies the necessary join conditions and also limits the rows that are fetched by specifying that the date of the sale (**cdate**) must fall between the report's start and end dates (**BETWEEN pr_ssum.sdate AND pr_ssum.edate**). The ORDER BY clause ensures the rows the SELECT fetches are in the proper order for the report (in employee number order).

The PROMPT statements display a message on the screen, wait for the user to enter something, and put the response in the specified variable (or record element). For example, the first PROMPT statement in Figure 7-5 displays Start Date (mm/dd/yy) : on the screen and puts the user's response in **pr_ssum.sdate**.

Although the declaration of the **c_sales** cursor makes use of the **sdate** and **edate** elements of the **pr_ssum** record, and the declaration appears in the code before these variables have been assigned values, the declaration is not evaluated until the FOREACH clause is executed. Thus the variables have been assigned values by the time they are used in the SELECT statement.

7. USING THE REPORT WRITER

```
DATABASE leads
GLOBALS "globals.4gl"

MAIN
{
The s_medium program computes the value of each sale for each
salesperson in a specified period and feeds them to the
r_medium report. The r_medium report prints the total value
of the sales for each employee as well as the grand total
for the period. The output is sent to medium.out.
}
DEFINE pr_ssum          RECORD
      tot              MONEY,
      emp             SMALLINT,
      sdate           DATE,
      edate           DATE
END RECORD

DECLARE c_sales CURSOR FOR
SELECT      sale.quantity * product.price * (1 - sale.discount/100),
            contact.empnum
      INTO   pr_ssum.tot,
            pr_ssum.emp
      FROM   sale,
            product,
            contact
      WHERE  sale.cnum = contact.cnum
            AND sale.pcode = product.pcode
            AND contact.cdate BETWEEN pr_ssum.sdate AND pr_ssum.edate
      ORDER BY contact.empnum

PROMPT "Start Date (mm/dd/yy): " FOR pr_ssum.sdate
PROMPT "End Date (mm/dd/yy): " FOR pr_ssum.edate
START REPORT r_medium
FOREACH c_sales
      OUTPUT TO REPORT r_medium(pr_ssum.*)
END FOREACH
FINISH REPORT r_medium
END MAIN
```

Figure 7-5: The Driver for the **r_medium** Report

The rest of the report driver starts the report, uses a **FOREACH** statement to loop through each of the rows that the **SELECT** fetches, and finishes the report.

The report itself is quite short, consisting mostly of definitions and the **PAGE HEADER** subsection. Refer to Figure 7-6. The **DEFINE** statement mimics that of the report driver, using the record name **rr**. The **OUTPUT** section specifies the report output is to go to a file, and the **PAGE HEADER** subsection displays a header that is appropriate to the report.

The heart of this report is the AFTER GROUP OF subsection. In order for this subsection to work properly, the report driver must pass the rows to the report in employee number order, which it does because of the ORDER BY clause in the SELECT statement (Figure 7-5). The report executes the statements in the AFTER GROUP OF subsection each time the specified variable (**rr.emp**) changes value. (While the statements in an AFTER GROUP OF subsection are executing, all variables still have the values from the last group. Thus, **rr.emp** will print as the employee number of the preceding group. You can think of the AFTER GROUP OF subsection as the “on last row of group” subsection.)

The GROUP SUM clause takes on the value of the sum of its argument over the entire group. In the case of this report, which is

```
REPORT r_medium(rr)
DEFINE      rr              RECORD
            tot             MONEY,
            emp             SMALLINT,
            sdate           DATE,
            edate           DATE
            END RECORD

OUTPUT
  REPORT TO "medium.out"

FORMAT
PAGE HEADER
  PRINT "Sales Summary by Salesperson"
  PRINT "For the Period ", rr.sdate, " through ", rr.edate
  SKIP 2 LINES
  PRINT "Employee", COLUMN 20, "Total"
  PRINT "Number", COLUMN 20, "Sales"
  SKIP 1 LINE

AFTER GROUP OF rr.emp
  PRINT rr.emp, COLUMN 15, GROUP SUM(rr.tot) USING "###,##&.&&"

ON LAST ROW
  SKIP 1 LINE
  PRINT "Total", COLUMN 13, SUM(rr.tot) USING "#,###,##&.&&"
END REPORT
```

Figure 7-6: The **r_medium** Report

grouping by salespeople, the group sum of **rr.tot** is the employee's total sales for the report period. The **USING** clause formats the output by inserting a comma if necessary and lining up the decimal points under the heading.

The **ON LAST ROW** subsection executes its statements after the driver has passed the report the last row of information. It is triggered by the **FINISH REPORT** statement in the report driver. The **SUM** clause is similar to the **GROUP SUM** clause except it sums its argument over all the rows of the report, not just one group.

A MORE COMPLEX REPORT

The **r_sales** report lists sales for a six-month period, by salesperson and month, with salesperson and month totals and a grand total. Chapter 2 sketched this report in Figure 2-3. Figure 7-7 shows the report output with 6/30/86 supplied as the end date.

S A L E S							
for the period Jan. 01, 1986 - Jun. 30, 1986							
by salesperson							
Salesperson	Jan	Feb	Mar	Apr	May	Jun	Total
J Bell	31	0	280	435	190	95	1,031
R Fuller	12	181	112	45	0	49	399
M Greenberg	372	130	0	170	213	723	1,608
Total	415	311	392	650	403	867	3,038

Press any key to continue. ■

Figure 7-7: Output of the **r_sales** Report

This discussion of the report driver and the report will focus on programming techniques that have not been covered previously. Refer to Appendix A for listings of the entire `rd_sales` report driver function and the `r_sales` report.

The Report Driver

The first order of business for the report driver `rd_sales` is to prompt the user for the end date of the report and figure out what months the report will span. The code shown in Figure 7-8 computes a start date based on the end date supplied by the user. If the user does not enter a date, the report uses `TODAY` as the end date. The start date is the first day of the month five months before the end date. The total span of the report period is between five and six months.

```
PROMPT "Enter end date for report as mm/dd/yy (or just RETURN for today): "  
    for edate  
DISPLAY "Running report, please wait." AT 15,1  
  
IF edate IS NULL THEN LET edate = TODAY END IF  
LET temp1 = MONTH(edate) - 5  
IF (temp1 <= 0) THEN  
    LET sdate = MDY(12 + temp1, "1", YEAR(edate) - 1)  
ELSE  
    IF (temp1 = 7) THEN {correct for wrong year: report run in Jan, end in Dec}  
        LET sdate = MDY(temp1, "1", YEAR(edate) - 1)  
    ELSE  
        LET sdate = MDY(temp1, "1", YEAR(edate))  
    END IF  
END IF  
LET smonth = MONTH(sdate)
```

Figure 7-8: Setting the Start and End Dates for the `rd_sales` Report

The code in Figure 7-8 uses the `MONTH` function to compute the month the report must start in. By subtracting 5 from the month the report ends in, you either get the month the report starts in or you get a value of 0 or less. A value of 0 or less indicates a month from the previous year; you add 12 to the value to get the month the report starts and you subtract 1 from the year the report ends in to get the year it starts.

An `IF` statement passes control to the appropriate `LET`

7. USING THE REPORT WRITER

statement to assign a value to **sdate**. The MDY function takes three arguments and returns a date value based on their representation of the month, day, and year. The LET statements use the MDY function to compute the start date. Finally, the start month is derived from the start date.

The next section of code, shown in Figure 7-9, contains the two SELECT statements that extract the necessary information from the database. The first of the two SELECTs extracts only the sales that fall within the report period and places them into a temporary table named **hold**. The INTO TEMP clause causes all the rows the SELECT fetches to go directly into the **hold** table. A DROP TABLE statement at the end of the report driver drops the table in case the user wants to run the report again during the same ses-

```
SELECT      sale.quantity * product.price * (1 - sale.discount/100) tot,
            MONTH(contact.cdate) mon,
            contact.empnum num
FROM        sale,
            contact,
            product
WHERE       sale.cnum = contact.cnum
            AND sale.pcode = product.pcode
            AND contact.cdate BETWEEN sdate AND edate
INTO TEMP   hold

DECLARE c_sale CURSOR FOR
SELECT      mon,
            num,
            SUM (tot),
            lname,
            fname
            INTO      mnth,
            emp,
            msales,
            dum1,
            dum2
FROM        hold,
            sperson
WHERE       empnum = num
GROUP BY    lname,
            fname,
            num,
            mon
ORDER BY    lname,
            fname
```

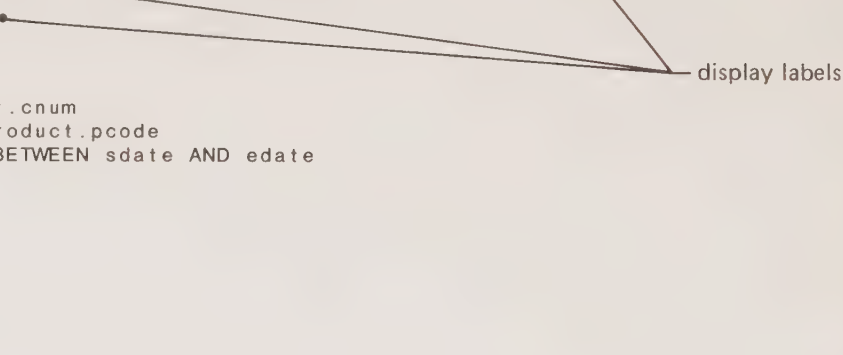


Figure 7-9: The SELECT Statements in **rd_sales**

sion. The second `SELECT` groups and sorts that information by salesperson and month.

The first `SELECT` performs a three-way join on each of the rows that satisfies the third criterion in the `WHERE` clause (the contact date must fall between the report's start date and end date). The first two criteria in the `WHERE` clause describe the relationships between the three tables to make sure only the desired rows are fetched.

The `hold` table has three columns—`tot`, `mon`, and `num`. The first expression following the first `SELECT` keyword is the value of the sale (quantity times price times the percent of full price paid). Because of the display label `tot` just before the comma at the end of the line, this value goes into `hold`'s `tot` column. The second expression makes use of the `MONTH` function to return the number representing the month of the year the sale occurred in. This expression goes into `mon`. The third value is the employee number and goes into `num`.

After this `SELECT` statement finishes execution, the `hold` table contains one row for each sale in the report period and nothing else. Although the second `SELECT` appears immediately after the first, it is actually only the definition of a cursor and is executed by the subsequent `FOREACH` statement. Refer to Figure 7-10.

When the second `SELECT` (Figure 7-9) is executed, it joins the `sperson` table with `hold` to obtain the names that are associated with the employee numbers. It also groups and orders by employee name and month. This grouping allows the use of the `SUM` aggregate function on the `tot` column to obtain the total sales for each group. Each group is a salesperson and a month, so you end up with the value of sales for a salesperson in a given month—just what was needed. The `SELECT` fetches rows in salesperson order.

```
START REPORT r_sales
FOREACH c_sale
  OUTPUT TO REPORT r_sales(sdate, edate, mnth, msales, emp)
END FOREACH
FINISH REPORT r_sales
```

Figure 7-10: The `FOREACH` Loop in `rd_sales`

7. USING THE REPORT WRITER

The FOREACH loop (Figure 7-10) feeds the report one row at a time from the second SELECT. After the final call to the report, the FINISH REPORT statement closes out the report, executing the statements following the ON LAST ROW subsection of the report.

The Report

The `r_sales` report, shown in Figure 7-11, starts with the REPORT statement followed by the DEFINE and OUTPUT sections. Although the report driver passes `msales` as a dollar value, the report receives it into a type INTEGER variable, avoiding possible discrepancies between the truncated, displayed values and their totals. The report uses the PAGE HEADER subsection of the FORMAT section to initialize the array variables that hold the names of the months (`months`) and the monthly totals (`totcol`). The initialization takes place only when PAGENO (an INFORMIX-4GL report variable that always contains the current page number of the report) is equal to 1. The PAGE HEADER subsection also clears the screen and prints the page header.

```
REPORT r_sales(rbegin, rend, mon, msales, emp)
{
  The r_sales report prints the value of sales for a six-month
  period, by salesperson and month, with salesperson and month
  totals and a grand total. The information is printed in
  tabular format on the screen.
}
DEFINE      msales          INTEGER,
            sfirst          CHAR(1),
            slast           CHAR(15),
            col, totcol     ARRAY[6] OF INTEGER,
            months          ARRAY[12] OF CHAR(3),
            rbegin, rend    DATE,
            mon, emp, temp1 SMALLINT

OUTPUT
  TOP MARGIN 0
  BOTTOM MARGIN 0
  LEFT MARGIN 0
  PAGE LENGTH 22

FORMAT

PAGE HEADER
  IF (PAGENO = 1) THEN
    LET months[1] = "Jan"   LET months[2] = "Feb"
    LET months[3] = "Mar"   LET months[4] = "Apr"
```



```

LET months[5] = "May"    LET months[6] = "Jun"
LET months[7] = "Jul"    LET months[8] = "Aug"
LET months[9] = "Sep"    LET months[10] = "Oct"
LET months[11] = "Nov"   LET months[12] = "Dec"
FOR temp1 = 1 TO 6
    LET totcol[temp1] = 0
END FOR
END IF

CLEAR SCREEN
PRINT COLUMN 35, "S A L E S"
PRINT COLUMN 20, "for the period ",
    rbegin USING "MMM. DD, YYYY", " - ", rend USING "MMM. DD, YYYY"
PRINT COLUMN 33, "by salesperson"
SKIP 1 LINE
LET temp1 = MONTH(rbegin)
PRINT "Salesperson", COLUMN 22, months[temp1];
LET temp1 = temp1 + 1
IF temp1 > 12 THEN LET temp1 = temp1 - 12 END IF
PRINT COLUMN 30, months[temp1];
LET temp1 = temp1 + 1
IF temp1 > 12 THEN LET temp1 = temp1 - 12 END IF
PRINT COLUMN 38, months[temp1];
LET temp1 = temp1 + 1
IF temp1 > 12 THEN LET temp1 = temp1 - 12 END IF
PRINT COLUMN 46, months[temp1];
LET temp1 = temp1 + 1
IF temp1 > 12 THEN LET temp1 = temp1 - 12 END IF
PRINT COLUMN 54, months[temp1];
LET temp1 = temp1 + 1
IF (temp1 > 12) THEN LET temp1 = temp1 - 12 END IF
PRINT COLUMN 62, months[temp1], COLUMN 74, "Total"
SKIP 1 LINE

```

Figure 7-11: The Beginning of the **r_sales** Report

The code in Figure 7-12 shows the USING clause that displays a formatted date.

```

PRINT COLUMN 20, "for the period ",
    rbegin USING "MMM. DD, YYYY", " - ", rend USING "MMM. DD, YYYY"

```

Figure 7-12: The PRINT USING Statement from the **r_sales** Report

The USING strings MMM. DD, YYYY in Figure 7-12 print the **rbegin** and **rend** date variables in the format you might expect: Jan. 15, 1986. You can vary the punctuation, the number of digits in the year (two or four), the type of representation of the month (use MM for a numeric representation), and the type of representation of the day (use DDD for a character representation of the day of the week). Refer to Figure 7-13.

Format String	Formatted Result
"mmdyy"	121385
"ddmmyy"	131285
"yymmdd"	851213
"yy/mm/dd"	85/12/13
"yy mm dd"	85 12 13
"yy-mm-dd"	85-12-13
"mmm. dd, yyyy"	Dec. 13, 1985
"mmm dd yyyy"	Dec 13 1985
"yyyy dd mm"	1985 13 12
"ddd, mmm. dd, yyyy"	Fri, Dec. 13, 1985
"(ddd) mmm. dd, yyyy"	(Fri) Dec. 13, 1985

Figure 7-13: Some Possible Date Formats with PRINT USING

Finally, the PAGE HEADER subsection (Figure 7-11) prints the appropriate list of months across the top of the screen, based on the beginning date of the report (**rbegin**).

The ON EVERY ROW subsection of the **r_sales** report (Figure 7-14) assigns the sales figures for the month (for a given employee) to the appropriate element of the **col** array. The index for the array is based on where the month fits in the report. For example, if the current row from **rd_sales** covers February (month 2) and the first month of the report is November (month 11), the sales figure should go in the fourth element of the array (February is the fourth month of the report). The computation takes into account the situation in which, as in the preceding example, the report spans a year end and the beginning month of the report is represented by a larger number than the current month.

```
ON EVERY ROW
  IF (MONTH(rbegin) > mon) THEN
    LET mon = mon + 12
  END IF
  LET mon = mon - MONTH(rbegin) + 1
  LET col[mon] = msales
```

Figure 7-14: Computing the Index for the Array

The BEFORE GROUP OF and AFTER GROUP OF subsections of `r_sales` (Figure 7-15) are executed each time the specified variable changes values. In order for these subsections to work, the report driver must supply rows that are grouped by the specified variable. The second SELECT in the `rd_sales` report driver (Figure 7-9) appropriately includes a GROUP BY clause that groups like employee numbers together. The BEFORE GROUP OF subsection sets all elements of the `col` array to 0.

```

BEFORE GROUP OF emp
  FOR temp1 = 1 TO 6
    LET col[temp1] = 0
  END FOR
AFTER GROUP OF emp
  SELECT      fname[1],
              lname
  INTO        sfirst,
              slast
  FROM        sperson
  WHERE       empnum = emp

  PRINT sfirst, 1 SPACE, slast,
  COLUMN 18, col[1] USING "###,##&",
  COLUMN 26, col[2] USING "###,##&",
  COLUMN 34, col[3] USING "###,##&",
  COLUMN 42, col[4] USING "###,##&",
  COLUMN 50, col[5] USING "###,##&",
  COLUMN 58, col[6] USING "###,##&",
  COLUMN 70, col[1] + col[2] + col[3] + col[4] + col[5] + col[6]
              USING "#,###,##&"

  FOR temp1 = 1 TO 6
    LET totcol[temp1] = totcol[temp1] + col[temp1]
  END FOR

PAGE TRAILER
  CALL wwait()

ON LAST ROW
  SKIP 1 LINE
  PRINT "Total",
  COLUMN 18, totcol[1] USING "###,##&",
  COLUMN 26, totcol[2] USING "###,##&",
  COLUMN 34, totcol[3] USING "###,##&",
  COLUMN 42, totcol[4] USING "###,##&",
  COLUMN 50, totcol[5] USING "###,##&",
  COLUMN 58, totcol[6] USING "###,##&",
  COLUMN 70, totcol[1] + totcol[2] + totcol[3]
              + totcol[4] + totcol[5] + totcol[6]
              USING "#,###,##&"

END REPORT

```

Figure 7-15: The End of the `r_sales` Report

The AFTER GROUP OF subsection uses a SELECT statement to fetch the current employee's first initial and last name, based on the **emp** variable. Then it uses a PRINT statement to display the employee's name and sales figures for the report period. All the values are printed with a single PRINT statement. Each element of the row is positioned using a COLUMN clause and formatted with a USING clause.

The USING clause ensures uniform spacing of the information, determines positions of commas, and forces a 0 to print in positions that would otherwise have no entry. The USING string uses pound signs (#) to represent digits that are printed as spaces if they are leading 0s, commas as themselves (a comma prints only if there is a digit to its left), and ampersands (&) to represent digits that are always printed (even if they are 0). Because the USING string does not specify a decimal point or any places to the right of the decimal point, it prints only whole numbers.

The PRINT statement prints the total column at the right of the screen by adding the values of all the columns in the row. The FOR loop at the end of the AFTER GROUP OF subsection keeps a running total of the sales in each month of the report.

The PAGE TRAILER subsection calls the **wwait** function, which causes the program to pause until the user presses a key. For multiscreen reports, this feature allows the user to read each screenful of information at his or her own pace before proceeding to the next screen.

When the report driver executes the FINISH REPORT statement, the report executes the statements in the ON LAST ROW subsection. In this report, this section prints the monthly totals and the grand total using the values that have been accumulated in the **totcol** array.

SUMMARY

Reports, as generated by a report writer, are critical to most 4GL applications. One of the main reasons someone uses a 4GL to

put data into a database is generally so that he or she can get out meaningful information in the form of reports. This chapter presented the basics of coding both reports and the programs that feed data to reports: the report drivers. It delved into both the formatting aspects of a report writer and the data manipulation features of a report writer. After reading the chapter you should be able to write reports based on your application using **INFORMIX-4GL**.

8

ADVANCED CONCEPTS AND FEATURES

This chapter explains how to use some of the most powerful features of **INFORMIX-4GL**. It describes the **qcontact**, **contact**, and **product** functions in detail, explaining how to use the

- **CONSTRUCT** and **PREPARE** statements to perform a query by example
- **DEFINE** statement to declare an array
- **BEFORE FIELD** and **AFTER FIELD** clauses of the **INPUT** statement to assist the user in entering data
- **DISPLAY ARRAY** statement to allow the user to choose from a list of items in a table
- **INPUT ARRAY** statement to make data handling easy for the novice and experienced user alike
- **DEFER INTERRUPT** statement to trap interrupts

QUERY BY EXAMPLE

A *query by example* is a query of a database in which the user supplies an example of what the row or rows he or she is looking for contain. The example row can include values, including wild-card characters, for one or more columns. The program then fetches and processes the rows the user specified by example.

When you choose **Update** from the **Top-Level** menu, and then choose **Contact** from the **Update** menu, the **qcontact** function displays the **f_qcontact** form. This form lets you specify, by example, the contact information you want to update.

Figure 8-1 shows the **f_qcontact** form after the user has filled it in. When the user presses **ESCAPE**, the query in Figure 8-1 will return all contacts after February 1, 1986, with a prospect with the last name of Pitts.

```
UPDATE CONTACT INFORMATION
To display the row for updating, fill in some fields, then press ESCAPE.
Once the row is displayed, update the appropriate fields, then press ESCAPE.

CONTACT INFORMATION:
      Date > 2/1/86
      Salesperson
      Next
Next salesperson
      Comments
      Type

PROSPECT INFORMATION:
      Last name Pitts ■
      Company name

SALES INFORMATION:
      Product code
      Quantity sold
      Discount
```

Figure 8-1: The **f_qcontact** Form After the User Has Filled It In

The process of building a query by example involves a **CONSTRUCT** statement that works with a form and a **PREPARE** statement that sets up a query for a cursor declaration (a **SELECT**

statement). To begin with, the INFORMIX-4GL program opens and displays a form on the screen (Figure 8-2). Then the CONSTRUCT statement allows the user to enter values in one or more of the fields in the form. These values communicate to the program what values, in which columns, the rows that the query fetches must contain. The user presses **ESCAPE** when he or she has finished constructing the query. The CONSTRUCT statement then takes the values the user entered, builds a WHERE clause from them, and puts this clause in a string. (If, as an experiment, you want to see the clause, you can use a DISPLAY statement to examine the string that CONSTRUCT builds.)

The next step is to assemble the entire query. A LET statement performs this task by concatenating several strings, including the

```

DATABASE leads
GLOBALS "globals.4gl"

FUNCTION qcontact()
{
  The qcontact function constructs a query by example based on
  the user's input in the f_qcontact form. It retrieves contacts
  that satisfy the query and displays each one. The contact the user
  selects to update is then updated in the contact table, and if
  the contact was a sale, in the sale table.
}
DEFINE      wbuf      CHAR(400),
            qbuf      CHAR(500),
            answer    CHAR(1),
            cnt       SMALLINT

CLEAR SCREEN
OPEN FORM f_qcontact FROM "f_qcontact"
DISPLAY FORM f_qcontact
CONSTRUCT wbuf ON
    contact.cdate,
    contact.empnnum,
    contact.ndate,
    contact.nemp,
    contact.notes,
    contact.ctype,
    prospect.lname,
    prospect.company,
    sale.pcode,
    sale.quantity,
    sale.discount
FROM sr_consale.*
  
```

Figure 8-2: The CONSTRUCT Statement from the qcontact Function

one **CONSTRUCT** built (Figure 8-3). Finally, a **PREPARE** statement compiles this string into a query you can use in a **DECLARE** statement.

The **CONSTRUCT** statement in the **qcontact** function builds its **WHERE** clause in the **wbuf** string variable. The **qbuf** variable holds the string that forms the entire query. After **qcontact** clears the screen and opens and displays the form, **CONSTRUCT** goes to work. Following the keyword **CONSTRUCT** is the name of the string variable into which **CONSTRUCT** will put the **WHERE** clause it builds (refer to Figure 8-2). Following the keyword **ON** is the list of all the columns that **CONSTRUCT** can use. The end of the **CONSTRUCT** statement is marked by the **FROM** clause that specifies the screen record that holds the fields that the user will enter values into. These fields must correspond to the preceding list of columns.

While entering values, the user can press the **ARROW** and **RETURN** keys to move the cursor. The user can also use wildcards and relational operators to accept a variety of values in a field. For example, entering a date of **> 9/30/85** in the **cdate** field will yield all rows that have a contact date of October 1, 1985, or later. Specifying *** mportant*** for the **notes** field will give you all rows that have Important, Unimportant, important, or unimportant in them, anywhere in the field.

After **CONSTRUCT** builds the **WHERE** clause in **wbuf**, a **LET** statement assembles the entire **SELECT** statement for the query. When **LET** concatenates strings, it does not put any space between the strings so it is up to you, the programmer, to include necessary spaces. See Figure 8-3.

```
LET qbuf =
  "SELECT      * ",
  "FROM        contact, prospect, OUTER sale ",
  "WHERE       sale.cnum = contact.cnum AND ",
              "prospect.ref = contact.ref AND ",
              wbuf CLIPPED
PREPARE q_1 FROM qbuf
DECLARE c_qcontact CURSOR FOR q_1
```

Figure 8-3: Setting Up and Preparing a Query

The `SELECT` statement that is built in Figure 8-3 includes all the necessary parts. It includes the keyword `WHERE`; the specification for a join between the three tables that participate in the query; and the keyword `AND`, which allows additional qualifiers to be added to the `WHERE` clause. The `wbuf` that `CONSTRUCT` built is tacked onto the end of the join specification in the `WHERE` clause to limit the query so it will fetch only the rows the user specified by example.

It is not necessary to break the `SELECT` statement into separate strings and then to concatenate them, as the example does; it is just easier to read that way. You could put the entire `SELECT` in a single string and simply append `wbuf` to it.

The keyword `OUTER` (for *outer join*) in the `SELECT` statement ensures that the query will return a set of values for a row in the join between the `contact` and `prospect` tables even if there is no corresponding row for it to join with in the `sale` table (this is the case when a contact is not a sale). Note the behavior of an outer join when you specify values for the query from both the `contact` (or `prospect`) and `sale` tables. Because any condition in the `WHERE` clause can be compromised if it depends on a table that is specified as `OUTER`, the `SELECT` will return rows from the `contact` and `prospect` tables even if the contact does not satisfy the criteria established for the `sale` table. It just will not display any values in the sales part of the form.

The `PREPARE` statement compiles the query named `q__1` from the string in the `qbuf` variable. Compiling the query creates executable code from an ASCII string. The `PREPARE` statement performs the same function as the `INFORMIX-4GL` compiler does when it sees a `SELECT` statement, except `PREPARE` does its work at run time.

The `qcontact` function does not introduce any new concepts after the `PREPARE` statement. The rest of the function (refer to Appendix A) uses a `FOREACH` loop to cycle through the rows the query returns, displaying each row on the form. A `PROMPT` following the display of each row asks the user if this is the row that needs to be updated. When the user chooses a row, the `qcontact` function executes an `INPUT BY NAME` statement and then updates the two tables in the database.

ARRAYS

The **contact** function uses arrays to assist the user in choosing a salesperson from the **sperson** table.

If the user does not know the employee number of the salesperson who made a contact, this function opens a window with a form and uses a **DISPLAY ARRAY** statement to display a list of names of salespeople. The user can scroll through this list until the cursor is on the name he or she wants to select. When the user presses **ESCAPE**, the window is closed, and the underlying form is restored with the employee number in the appropriate field on the form.

This function works by keeping two arrays—one filled with employee numbers, the other filled with the corresponding employee names. When the user chooses a salesperson from the list on the screen, the function notes the index of that salesperson in the array of employee names. Then it assigns the corresponding element of the employee number array to the appropriate variable and displays it on the **f_contact** form.

The following sections show pieces of code from the **contact** function. Refer to Appendix A for a complete listing.

Defining an Array

Before you can use a **DISPLAY ARRAY** statement to display information on a form, you must define the array and fill it with the information you want to display. Figure 8-4 shows the definition of the **pa_empnum** (employee number) and **pa_spers** (salesperson name) arrays in the **contact** function.

```
DEFINE    pa_empnum          ARRAY[25] OF LIKE sperson.empnum,  
          pa_spers          ARRAY[25] OF CHAR(31),
```

Figure 8-4: Defining Arrays in the **contact** Function

The keyword `ARRAY` declares these variables to be of type `ARRAY`. The number in square brackets specifies the number of elements within the array. In the example, both arrays have 25 elements. Because of the `LIKE` clause, each of the elements in the `pa_empnum` array is of the same type as the `empnum` column in the `sperson` table. The `pa_spers` array has elements that are of type `CHAR` and are 31 characters long (allowing 15 characters each for the first and last names and one space between them).

An array differs from a record in that in an array all the elements are the same, whereas a record contains a set of (usually) dissimilar elements. As you will see in the discussion of the `product` function, the elements of an array can be records. However, you cannot define a record to have arrays as elements.

Assigning Values to an Array

Once the arrays have been declared, the next order of business is to fill them with the appropriate information from the database. The statements shown in Figure 8-5 fill the `pa_empnum` array with information from the `sperson` table.

```
DECLARE c_sperson CURSOR FOR
  SELECT      fname,
              lname,
              empnum
  FROM        sperson
  ORDER BY    lname,
              fname

LET scount = 1
FOREACH c_sperson
  INTO      buf1,
            buf2,
            pa_empnum[scount]
  LET pa_spers[scount] = buf1 CLIPPED, 1 SPACE, buf2
  LET scount = scount + 1
END FOREACH
```

Figure 8-5: Filling an Array in the `contact` Function

The **c_sperson** cursor and its associated **SELECT** statement fetch the first name, last name, and employee number from the **sperson** table. The **ORDER BY** clause ensures that the **SELECT** fetches the names in alphabetical order (so the user will be presented with an alphabetical list). This particular **SELECT** statement *does not* specify where the results are to go; that is left up to the **FOREACH** statement.

The statement before the start of the **FOREACH** loop sets the index **scount** to 1, the index of the first element of the array. The **INTO** clause of the **FOREACH** statement puts the first name into the variable **buf1**, the last name into **buf2**, and the employee number directly into the element of the **pa_empnum** array specified by **scount**. The subsequent **LET** statement assigns the value of the first and last names, with one space in between, to the element of **pa_spers** specified by **scount**. Finally, the routine increments **scount**. After the **FOREACH** loop goes through all the rows in the table, **scount** has a value of one more than the number of rows in the table. Later in the program, one is subtracted from the index before it is used.

Now the array is ready for action, but you need a form to display it on. Figure 5-13 shows the form, **f_empnum**, that is used to display this array.

Setting Up the Form

Each field in a form must be associated with a column in a table. The pseudotable **FORMONLY** provides a way of putting fields in a form that do not correspond to any column. In Figure 5-13 the names of the salespeople will appear in the fields that contain the tags **c06**. Because these fields do not correspond directly to any column in the table, they are declared to belong to **FORMONLY**.

The **INSTRUCTIONS** section of the form shown in Figure 5-13 specifies that the delimiters are to be spaces (there are two spaces between the quotation marks following the keyword **DELIMITERS**). If you do not specify delimiters, each of the fields on the form is delimited by square brackets. Spaces rather than square brackets are used as delimiters on the **f_empnum** form because there is no

need to emphasize the boundaries of the fields and spaces make the screen appear less cluttered. Square brackets or other visible characters are preferable when either the location or length of a field is ambiguous. To illustrate the alternatives, some of the forms in the **leads** application were designed with square brackets whereas others use spaces as delimiters. Some use left-justified labels whereas others use right-justified labels to make it obvious where the blank-delimited fields start (refer to Figure 8-1). Although the forms in the various **leads** programs use a variety of combinations of delimiters and field labels, it is best to establish a consistent policy and follow it throughout an application.

The second statement in the **INSTRUCTIONS** section defines the screen record that the **INFORMIX-4GL** program can refer to. The name of the screen record is **sr_con1**, it contains five rows (this number must correspond to the number of rows in the form that contain the tag or tags for the fields in the record), and each row contains the field or fields specified in the parentheses (**FORMONLY.fname** in the example).

The BEFORE FIELD and AFTER FIELD Clauses

The first part of the **INPUT** statement from the **contact** function is shown in Figure 8-6.

```
LET msg2 = "Enter 0 to display and choose from a list of values."
.
.
.
INPUT BY NAME  pr_contact.cdate THRU pr_contact.ctype

    AFTER FIELD cdate
        LET pr_contact.ndate = pr_contact.cdate + 14
        DISPLAY BY NAME pr_contact.ndate

    BEFORE FIELD empnum
        MESSAGE msg2 ATTRIBUTE(REVERSE)
.
.
.
END INPUT
```

Figure 8-6: The **INPUT BY NAME** Statement from the **contact** Function

The `INPUT BY NAME` statement allows the user to enter values into variables. In Figure 8-6, the specified variables are parts of the `pr_contact` record (defined in the `globals.4gl` file). Because this file defines the record to be like a table in the database, you must refer back to the `create` program to find out exactly which variables this statement will accept values into. (You can also look at the `f_contact.per` form definition file, but if this file is missing or incorrect, the final authority for the list could be traced back to the schema in `globals.4gl`.)

The `INPUT` statement does not end until the `END INPUT` clause. The bulk of the statement is taken up with instructions that tell `INFORMIX-4GL` what to do before and after the user moves the cursor into different fields. `INFORMIX-4GL` executes the first `AFTER FIELD` clause (shown in Figure 8-6) after the user moves the cursor out of the contact date (`cdate`) field. This `AFTER FIELD` clause executes two statements. The first assigns the value of the date that is 14 days past the contact date to the next date (`ndate`) field. The second statement displays this date in the next date field on the form.

Next is a `BEFORE FIELD` clause that is executed just as the user moves the cursor to the employee number (`empnum`) field. This clause executes a `MESSAGE` statement that displays the contents of the `msg2` variable on the message line (normally the second line of the form) in reverse video (dark letters on a light background).

The next `AFTER FIELD` clause, shown in Figure 8-7, also works with the employee number field, but it is not executed until the user moves the cursor out of the field. This `AFTER FIELD` clause helps the user if he or she does not know the correct employee number to enter in the field or if the user enters an invalid employee number. In either of these cases, the `AFTER FIELD empnum` clause opens a window, displays a form, and displays a list of employee names for the user to choose from. As the next sections discuss, if the user does not enter anything (null), enters a 0, or enters an invalid employee number, the function opens a window and displays the `pa_spers` array, which is filled with salespeople's names. The only way the user can leave this field without getting assistance in selecting an employee is by entering a valid employee number in the field.

```

AFTER FIELD empnum
  LET eflag = -1
  IF (pr_contact.empnum IS NOT NULL
    AND pr_contact.empnum != 0) THEN
    SELECT      COUNT(*)
      INTO      cnt
    FROM        sperson
    WHERE       empnum = pr_contact.empnum
    IF (cnt != 1) THEN
      ERROR "There is no salesperson number ",
        pr_contact.empnum USING "###"
    ELSE
      LET eflag = 0
    END IF
  END IF

```

Figure 8-7: An AFTER FIELD Statement from the **contact** Function

The first thing the AFTER FIELD clause in Figure 8-7 does is to set **eflag** to -1. The **eflag** variable is used to track the user's performance. If the **eflag** variable has a value that is less than 0 when it is tested, it means the user did not enter a valid employee number in the field.

The first IF statement checks to see if the user entered a value other than 0 or null. If the user did, a SELECT statement counts the number of employees with the employee number the user specified. If the count is not equal to 1, it means the user did not enter a valid employee number and an error message is displayed. If it is equal to 1, **eflag** is reset to 0. The next IF (Figure 8-8) tests **eflag** to see if the user entered a valid employee number.

The DISPLAY ARRAY Statement

The INFORMIX-4GL SET_COUNT function works with both DISPLAY ARRAY and INPUT ARRAY statements. You use it to specify the number of rows of data the array actually contains—not the dimension of the array. After determining that the user did not enter a valid employee number, the **contact** function resets the error flag to 0 and calls the SET_COUNT function with the number of rows in the array. Refer to Figure 8-8. (The **scount** variable was set in the FOREACH loop that filled the **pa_spers** array. Because of the way it was incremented, it ended up having a value one more than the number of rows in the array. Thus, 1 is subtracted from it in this assignment statement.)

8. ADVANCED CONCEPTS AND FEATURES

```
LET msg1 = "Use ARROW keys to move highlight, ESCAPE to make selection."
.
.
.
IF (eflag < 0) THEN
  LET eflag = 0
  CALL SET_COUNT(scount - 1)
  OPEN WINDOW w_empnum AT 6,10 WITH FORM "f_empnum" ATTRIBUTE(BORDER)
  MESSAGE msg1 ATTRIBUTE(REVERSE)
  DISPLAY ARRAY pa_spers TO sr_con1.*
  CLOSE WINDOW w_empnum
  LET cnt = ARR_CURR()
  LET pr_contact.empnum = pa_empnum[cnt]
  DISPLAY BY NAME pr_contact.empnum
END IF
LET pr_contact.nemp = pr_contact.empnum
DISPLAY BY NAME pr_contact.nemp
```

Figure 8-8: Using the DISPLAY ARRAY Statement

The **contact** function then opens the **w_empnum** window and displays the **f_empnum** form. After displaying another message to help the user (**msg1**), the function executes the **DISPLAY ARRAY** statement. This statement displays the **pa_spers** array to the **sr_con1** screen record.

After displaying as many rows from the array as the screen record will permit (five in the case of the example), the **DISPLAY ARRAY** statement leaves the cursor on the first row. The message at the top of the window tells the user to use the **ARROW** keys to move the cursor until it is on the correct salesperson and then to press **ESCAPE** (Figure 5-11).

Now the user who has asked for a list of employees or has entered an invalid employee number has a list of employees displayed in a window on the screen. At this point the **ARROW** and **RETURN** keys will scroll the array through the screen record as necessary. When the cursor is on the bottom row of the screen record (the fifth line in the example) and the user presses the **DOWN ARROW** key, **INFORMIX-4GL** scrolls the display to reveal a new row from the program array at the bottom of the screen record and forces the top row off the screen. The **UP ARROW** key works in a similar manner when the cursor is at the top of the screen record. In addition, you can use function keys **F3** and **F4** to fast forward and fast reverse through the array.

As soon as the user presses **ESCAPE**, the program executes a **CLOSE WINDOW** statement that removes the window and restores the **f_contact** form. Refer back to Figure 8-8. After execution of a **DISPLAY ARRAY** statement, you can tell which element of the array the highlight was on when the user pressed **ESCAPE** by calling the **INFORMIX-4GL ARR_CURR** function. In the example, the value returned by this function is assigned to **cnt**. The next statement works because of the way the two arrays were set up. The **cnt** variable contains the index of the element of the **pa_spers** array that contains the salesperson the user chose. The corresponding element of the **pa_empnum** array contains the employee number of the same salesperson. The following statements assign this value to the **empnum** element of the **pr_contact** record and display the value on the **f_contact** form.

The final two statements assign the value the user just chose to the **nemp** element of the record and display that value.

When the user presses **ESCAPE** to choose a salesperson, all the user sees is an employee number appearing on the form in two places: the salesperson field and the next salesperson field.

The preceding code (Figure 8-7) is set up to accept 0 as a request to display the list of salespeople to avoid the problem created when an employee number is already in the field and the user wants to see the list. This situation arises when the user returns to the field to change it after entering a value. It is also always the situation in the next employee field, which is automatically filled out when the user enters the initial employee number. If a null was the only valid request to see the list of salespeople, the user would have to clear the field manually by entering **SPACES** or pressing **CONTROL-D** before pressing **RETURN** in order to see the list. The way the program is set up, the user knows that he or she can always enter 0 to see the list. Because null still works, the experienced user can just press **RETURN** without entering a 0 to see the list when the field is empty.

The **INPUT** statement continues on for quite a while, checking the data the user enters and assisting the user where possible. The **END INPUT** clause is followed by an **IF** statement that tests to see whether the user entered an **S** in the **ctype** field. If the user entered an **S**, the **w_sale** window is opened, the **f_sale** form is

displayed, and another INPUT statement begins. Thus the user can enter values in the sales window only if the contact was a sale. When the user is entering data about a sale and needs help with a product code, the **contact** function opens another window and uses another DISPLAY ARRAY statement to list the products. Finally, the END INPUT clause ends the second INPUT statement.

Putting the Data into the Table

The rest of the program adds the data the user entered to the appropriate tables. Refer to Figure 8-9.

```
INSERT INTO contact VALUES (pr_contact.*)
IF (pr_contact.ctype = "S") THEN
    LET pr_sale.cnum = SQLCA.SQLERRD[2]
    INSERT INTO sale VALUES (pr_sale.*)
END IF

CLEAR SCREEN
END FUNCTION
```

Figure 8-9: The INSERT Statements from the **contact** Function

The first INSERT statement adds a row to the **contact** table. Then an IF statement tests to see if the contact was a sale (if it was a sale, the **sale** table needs to have a row describing the sale added to it). Before the **pr_sale** record is complete, it must contain the contact number. But the contact number comes from the **contact** table. Because it is a type SERIAL column, its value is assigned by **INFORMIX-4GL** at the time the row is added to the table. Following the INSERT, the value that **INFORMIX-4GL** assigned is available as `SQLCA.SQLERRD[2]`. Figure 8-9 shows how this element of an **INFORMIX-4GL** array is used to put the serial number in the **sale** table.

INPUT ARRAY

The INPUT ARRAY statement works similarly to the way the DISPLAY ARRAY statement works except it allows you to enter or change the data in the array instead of just looking at it. This section builds upon the earlier part of this chapter and uses the **product** function to illustrate the INPUT ARRAY statement. Refer to Figure 8-10 for an example of what the **f_listprod** form looks like on the screen. Appendix A contains a listing of the **product** function in its entirety.

PRODUCT INFORMATION:

Use ARROW keys and RETURN to move between fields and rows.
 Press FUNCTION KEY 1 to insert a new row.
 Press FUNCTION KEY 2 to delete a row.
 Press CONTROL-B before you leave a row to change it back to the way it was.
 Press ESCAPE to exit.

Code	Price	Description
10d	\$1.55	1.0" D-ring (special order)
10r	\$1.35	1.0" regular binder
15d	\$1.97	1.5" D-ring binder
20d	\$2.32	2.0" D-ring binder
25d	\$2.65	2.5" D-ring binder

Figure 8-10: The **f_listprod** Form on the Screen

Although the INPUT ARRAY statement in the **product** function is quite complex and contains many statements, some INPUT ARRAY statements are simpler and take up only a few lines of code (see Figure 1-22). The INPUT ARRAY statement in the **product** function is complex and lengthy because it does a lot of work. Not only does it accept input from the user, but it also does error checking and manipulates the data in the **product** table.

The **product** table has three columns in it: **pcode**, **price**, and

descrip. In order to accommodate all three fields, the **product** function uses an array of records. Each record has three elements in it, one corresponding to each of the columns in the table. There are 20 records in the array. (If you have more than 20 products, you will need to define a larger array.) The part of the **DEFINE** statement that declares the array is shown in Figure 8-11.

```
DEFINE    pa_prod                                ARRAY[20] OF RECORD LIKE product.*,
```

Figure 8-11: Defining an Array of Records in the **product** Function

This example uses almost the same technique for filling the array as the previous example used. One difference is that in the current example the **SELECT** statement includes the **INTO** clause. Refer to Figure 8-12. As a result, the index in the **FOREACH** loop is correct after the loop finishes execution. Another difference is in the assignment statement in the **FOREACH** loop.

```
DECLARE c_prod CURSOR FOR
SELECT      *
      INTO   pr_product.*
      FROM   product
      ORDER BY pcode
LET idx = 0
FOREACH c_prod
  LET idx = idx + 1
  LET pa_prod[idx].* = pr_product.*
END FOREACH
CALL SET_COUNT(idx)
```

Figure 8-12: Filling an Array of Records from the **product** Function

The **SELECT** statement fetches all columns from all rows of the **product** table, ordered by the value of the **pcode** column. The **FOREACH** loop assigns the values of each of the elements of the **pr_product** record to each of the elements of one of the records that is in the **pa_prod** array. The index of the record in the array is determined by the **idx** variable. Following the loop, the **INFORMIX-4GL SET_COUNT** function sets things up for the **INPUT ARRAY** statement.

Like the **DISPLAY ARRAY** statement, the **INPUT ARRAY** statement allows the user to move the cursor through the array. When the

user presses the **ARROW** and function keys, **INFORMIX-4GL** scrolls the array through the screen records as needed so the user can view all elements of the array.

In addition, the **INPUT ARRAY** statement lets the user change data in the array by entering a value over an existing one. It also lets the user add data by pressing the **F1** function key and entering a new row of data. (The user can also use the empty row past the last row of the array for adding a new row.) Rows of data can be deleted by pressing the **F2** function key. The **F3** and **F4** function keys scroll the array forward and backward by pages. You can use the **OPTIONS** statement to assign these functions to other keys—refer to the **INFORMIX-4GL Reference Manual** for more information.

These insert, delete, and change features affect only **pa__prod** and the screen arrays; they do not, by themselves, make any changes to the **pr__product** record or to the database table. The **product** function uses several of the clauses of the **INPUT ARRAY** statement to pass the changes the user makes to the array on to the **product** table. When the user deletes a row from the screen (and from the array), the **product** function deletes the row from the table. In a similar manner, when the user adds or changes a row, the function makes the corresponding changes to the table.

The **INPUT ARRAY** statement supplies a variety of clauses that allow you (the programmer) to keep close tabs on what the user is doing. It executes the statements in the **AFTER FIELD** clause as the user moves the cursor out of the named field. It executes the **BEFORE ROW** and **AFTER ROW** clauses as the user moves the cursor from one row to another. The **BEFORE INSERT** and **BEFORE DELETE** are executed before, and the **AFTER INSERT** and **AFTER DELETE** are executed after, the user adds a row to the array and deletes a row from the array. (The **BEFORE ROW** clause is always executed before the **BEFORE INSERT** and **BEFORE DELETE** clauses, whereas the **AFTER ROW** clause is executed after the **AFTER INSERT** and **AFTER DELETE** clauses.)

The **product** function is based entirely on an **INPUT ARRAY** statement. The code that implements this statement will be discussed after the following overview. The **product** function uses these clauses within the **INPUT ARRAY** statement:

BEFORE ROW	Initializes indexes. Saves existing values from the current row in pr_product.* . Resets the insert/delete flag (iflag).
AFTER DELETE	Sets the insert/delete flag. Deletes the current row from the product table.
ON KEY (CONTROL-B)	Restores the row to the way it was before the user moved the cursor onto it (the row is restored from pr_product.*).
BEFORE INSERT	Initializes pr_product.* to nulls in case the user presses CONTROL-B. Without this initialization, the function would display values from the previous row if the user pressed CONTROL-B during an insert.
AFTER FIELD	Three AFTER FIELD clauses force the user to enter a value into each field. They do, however, allow the user to move the cursor from a field if nothing has yet been entered into the row. The AFTER FIELD clause for the pcode field also checks that the new product code is unique.
AFTER INSERT	Sets the insert/delete flag. Inserts the current row into the product table.
AFTER ROW	If the insert/delete flag has not been set, checks to see if anything in the row has been changed (by comparing it to pr_product.*), and if it has, updates the current row in the product table.

With the preceding brief description as a starting point, the following paragraphs describe the **product** function code in detail.

Figure 8-13 shows the beginning of the INPUT ARRAY statement from the **product** function, including the BEFORE ROW clause. The INPUT ARRAY statement uses a WITHOUT DEFAULTS clause. This clause causes the INPUT ARRAY statement to display the current values of the elements of the records that make up the array into which it is accepting values. Because this function gives the user the opportunity to change, add to, and delete data from the table,

the user needs to see the current values. The `BEFORE ROW` clause in Figure 8-13 initializes program variables each time the user moves the cursor to a new row. (It also initializes them when `INPUT ARRAY` gets going, before the cursor appears on the first row.)

```
INPUT ARRAY pa_prod WITHOUT DEFAULTS FROM sr_product.*
BEFORE ROW
  LET idx = ARR_CURR()
  LET scrn = SCR_LINE()
  LET pr_product.* = pa_prod[idx].*
  LET iflag = 0
```

Figure 8-13: The `BEFORE ROW` Clause from the `product` Function

The `idx` variable holds the value of the index of the current row in the array (the one the cursor is on) as returned by the `ARR_CURR` function. The `scrn` variable holds the value of the index of the current row in the `screen` array (the one the cursor is on) as returned by the `SCR_LINE` function. Both the `ARR_CURR` and `SCR_LINE` functions are supplied with `INFORMIX-4GL`; they are not user functions. Because it will be necessary to check to see if the user changed any of the elements of the array, the `pr_product` record is assigned values from the current element of the array. Later, each element of this record will be compared with the corresponding element of the current row in the array to determine if the `product` table needs to be updated.

Finally, the `BEFORE ROW` clause sets the `iflag` (insert/delete flag) to 0. This flag keeps track of whether the current row has been deleted, inserted, or has an error on it. In any of these cases, the flag is set to -1 and the `AFTER ROW` clause will not check to see if the row in the table needs to be updated.

`INFORMIX-4GL` executes the statements following the `ON KEY` clause when the user presses `CONTROL-B` (Figure 8-14). (You can use any key in place of `CONTROL-B` except `DEL` [or other interrupt key], `CONTROL-A`, `CONTROL-D`, `CONTROL-H`, `CONTROL-L`, `CONTROL-R`, or `CONTROL-X`. You can even use the function keys by using a clause such as `ON KEY (F7)`.)

```
ON KEY (CONTROL-B)
  LET pa_prod[idx].* = pr_product.*
  DISPLAY pa_prod[idx].* TO sr_product[scrn].*
  NEXT FIELD pcode
```

Figure 8-14: The ON KEY Clause from the **product** Function

In the case of the **product** function, pressing **CONTROL-B** returns the row to the state it was in before the user moved the cursor to it—**CONTROL-B** is an undo key. The first thing that happens when control is passed to this clause is the current record of the array is assigned values from the **pr_product** record, returning the current record of the array to its original state. Next, the current record of the array is redisplayed to the current element of the screen record, returning the fields in the current screen record to their previous values. Finally, **pcode** is specified as the next field the cursor will appear on.

If you do not specify the first field in the row as the next field following an ON KEY clause, **INFORMIX-4GL** maintains the buffer that was accumulating the user's keystrokes. When the user moves the cursor from the row, **INFORMIX-4GL** will reset both the display and the array to what this buffer contains, even though the program changed both of them. By specifying the first field in the row as the next field, you clear the buffer that accumulates the user's keystrokes and preserve your changes to the screen record and the array.

Following the ON KEY clause, an AFTER FIELD clause makes sure that, if the user has entered a value in any field in the current row, he or she enters a value in the **pcode** field (Figure 8-15). Then, if the user has entered a value, it checks to see that the value does not duplicate a value already in the table. (This technique of checking for duplicate values works fine in this application where it is assumed that there is only one database administrator who would be using this function. Other measures would have to be taken to ensure data integrity in an environment where more than one person is updating a table at one time.)

```

AFTER FIELD pcode
  IF (pa_prod[idx].pcode IS NULL) THEN
    IF (pa_prod[idx].price IS NOT NULL
        OR pa_prod[idx].descrip IS NOT NULL) THEN
      ERROR "You must enter a product code."
    NEXT FIELD pcode
  END IF
ELSE
  IF (pa_prod[idx].pcode != pr_product.pcode
      OR pr_product.pcode IS NULL) THEN
    SELECT      COUNT(*)
      INTO      cnt
      FROM      product
      WHERE     pcode = pa_prod[idx].pcode
    IF (cnt != 0) THEN
      ERROR "Product code must be unique."
    NEXT FIELD pcode
  END IF
END IF
END IF
```

Figure 8-15: The AFTER FIELD **pcode** Clause from the **product** Function

The initial IF statement in Figure 8-15 checks to see if the user entered a value in the product code field. If the user just pressed RETURN or used the ARROW keys to move the cursor from the field, **pa_prod[idx].pcode** will be null. If this is the case, the **product** function checks to see if the other two fields in the same row are also null. If they are, the row is empty and no further checking is required. The special case of an all-null row is handled by the AFTER ROW clause. If either of the other fields is not null, the cursor is returned to the product code field and the user is forced to enter something. (The user can always move the cursor off the row without entering a value by pressing CONTROL-B to return the row to its previous state or by pressing function key F2 to delete the row.)

If **pa_prod[idx].pcode** is not null, the next IF checks to see if the user changed the original entry in this field. If the user did make a change to it, then **pa_prod[idx].pcode** will not be equal to **pr_product.pcode**. To cover the case where the user is inserting a new row into the table, the ELSE IF also tests to see if **pr_product.pcode** is null.

Because of the way nulls work in Boolean expressions, the test

for a null value must be made explicitly. If the variable **pr__product.pcode** is null, the expression

```
pa__prod[idx].pcode != pr__product.pcode
```

will return a false value, regardless of the value of the left side of the expression. Because a null represents an unknown value, you can never know that it is not equal to another value, even another null value. You can only test it to see if it is equal to NULL or NOT NULL. Refer to Appendix B for more information on nulls.

If the user has changed the value in the product code field, the AFTER FIELD clause tests to see that the value is unique by using the COUNT aggregate to check that there are no rows with the same value in the product code column. If **cnt** takes on a value other than 0, it means the value the user entered is not unique. An error message is displayed, and the user is forced to enter another value. If the user wants to leave the row without entering a value, he or she can always press CONTROL-B to return the row to the way it was or press F2 to delete the row.

The next two AFTER FIELD clauses (not shown) work similarly to the way the preceding one works except they do not test for a unique value in a column.

The BEFORE INSERT clause (Figure 8-16) initializes the **pr__product** record to nulls. Without this clause, if the user is inserting a new row by pressing the F1 function key and presses CONTROL-B, the row will be returned to the value of the row that the cursor was previously on. With this clause, each time a row is being inserted, the “old value” of the row is set to nulls.

The AFTER INSERT clause (also Figure 8-16) sets the **iflag** to -1 so the AFTER ROW clause will not process the row, and it then attempts to insert the row into the **product** table. The first IF statement checks to see that the product code is not null. If it is null, the ELSE clause sets the error flag to -1 and no attempt is made to insert the row.

If the product code is not null, the function executes a WHENEVER ERROR CONTINUE statement, attempts to insert the row into the **product** table, and tests the **INFORMIX-4GL status** variable to see if the insert was successful. The WHENEVER ERROR CONTINUE


```
BEFORE INSERT
  INITIALIZE pr_product.* TO NULL

AFTER INSERT
  LET iflag = -1
  LET eflag = 0
  IF (pa_prod[idx].pcode IS NOT NULL) THEN
    WHENEVER ERROR CONTINUE
    INSERT INTO product VALUES (pa_prod[idx].*)
    IF (status < 0) THEN LET eflag = -1 END IF
    WHENEVER ERROR STOP
  ELSE
    LET eflag = -1
  END IF
  IF (eflag < 0) THEN
    ERROR "An error has occurred. Please enter the information again."
    INITIALIZE pa_prod[idx].* TO NULL
    CLEAR sr_product[scrn].*
    LET eflag = 0
  END IF
```

Figure 8-16: The AFTER INSERT Clause from the **product** Function

statement causes **INFORMIX-4GL** to continue executing the program even if the **INSERT** statement fails. Without this statement **INFORMIX-4GL** would display an error message and return control to the operating system if the **INSERT** statement failed. The **WHENEVER ERROR STOP** statement returns things to the way they were: **INFORMIX-4GL** will abort program execution if it encounters an error.

The **INSERT** statement inserts into the **product** table the values the user entered into the record of the **pa_prod** array specified by the index **idx**. The following **IF** statement tests the value of the **status** variable to see if an error occurred (a negative value indicates an error). If an error occurs (such as a duplicate product code), the function displays an error message, initializes the current record of the **pa_prod** array, and clears the current row on the screen. The user must reenter the row from the start. If you want to do more extensive error testing and reporting, you can save the status by assigning the value of the **status** variable to a program variable immediately after the **INSERT** statement and then taking action based on the value of this variable.

The **AFTER DELETE** clause (Figure 8-17) sets **iflag** and deletes the row from the **product** table where the **pcode** column is equal to

8. ADVANCED CONCEPTS AND FEATURES

the value of the **pcode** element of the **pr_product** record. Because the **pr_product** record still holds the value (of the current row) it was assigned in the BEFORE ROW clause, the program deletes from the **product** table the row corresponding to the row the cursor is on.

```
AFTER DELETE
  LET iflag = -1
  DELETE FROM product WHERE pcode = pr_product.pcode
```

Figure 8-17: The AFTER DELETE Clause from the **product** Function

The AFTER ROW clause shown in Figure 8-18 is the last clause INFORMIX-4GL executes as the user moves the cursor from a row. In the **product** function, this clause takes care of situations that the other clauses could not handle.

```
AFTER ROW
{All nulls, set flag to ignore row.}
IF (pa_prod[idx].pcode IS NULL
  AND pa_prod[idx].price IS NULL
  AND pa_prod[idx].descrip IS NULL) THEN
  LET iflag = -1
END IF

{User made a change to the row: update.}
IF (iflag = 0
  AND (pr_product.pcode != pa_prod[idx].pcode
    OR pr_product.price != pa_prod[idx].price
    OR pr_product.descrip != pa_prod[idx].descrip)) THEN
  UPDATE product SET
    product.* = pa_prod[idx].*
  WHERE pcode = pr_product.pcode
END IF

{User entered new data in a previously null row: insert.}
IF (iflag = 0
  AND (pa_prod[idx].pcode IS NOT NULL
    AND pr_product.pcode IS NULL)) THEN
  WHENEVER ERROR CONTINUE
  INSERT INTO product VALUES (pa_prod[idx].*)
  IF (status < 0) THEN
    ERROR "An error has occurred. Please enter the information again."
    INITIALIZE pa_prod[idx].* TO NULL
    CLEAR sr_product[scrn].*
  END IF
  WHENEVER ERROR STOP
END IF
```

Figure 8-18: The AFTER ROW Clause from the **product** Function

The first IF statement in the AFTER ROW clause checks to see if the row contains all null values. If the row does contain all nulls, it sets the **iflag** variable to -1 so that the rest of the clause will not process the row.

The second IF checks that the **iflag** is still 0 (indicating the row has not been processed by another clause and that it does not contain all nulls) and tests to see if the user has changed any one of the values in the row. It performs this test by comparing the current values of the fields (in the **pa_prod** array) with the values the fields had when the user first moved the cursor to the row (in the **pr_product** record). If **iflag** is 0 and a value has been changed, the function updates the row.

The third IF checks that **iflag** is still 0 and tests to see if the product code field was null when the user moved the cursor to the row and if it now contains a value. (This situation can arise, and not be taken care of by the AFTER INSERT clause, when the user starts but does not complete an insert and then moves the cursor off the row. The **product** function leaves a blank row on the screen. If the user moves the cursor back to this row and adds information, **INFORMIX-4GL** does not consider it an *insert* because the user is not inserting a row into the array and it does not execute the AFTER INSERT clause. But as far as the way this function works, the user is inserting a row into the table and the function must handle it as such.) If these conditions are true, the function attempts to insert the row into the table.

TRAPPING INTERRUPTS

If you do not trap interrupts and the user presses the DEL key (or whatever key your system uses to interrupt processing) while running an **INFORMIX-4GL** program, **INFORMIX-4GL** stops executing the program and returns control to the operating system. You can use a DEFER INTERRUPT statement to trap interrupts. A program can have only one of these statements, and it must appear in the

8. ADVANCED CONCEPTS AND FEATURES

main program (not a function). After INFORMIX-4GL executes a DEFER INTERRUPT statement, it will not return control to the operating system when the user presses DEL but will set the INFORMIX-4GL global variable `int_flag` to a nonzero value. If the user presses DEL during an INPUT or PROMPT statement, INFORMIX-4GL will pass control out of the input statement and set `int_flag`. It is up to you, the programmer, to check and reset the value of this variable periodically in the program and take appropriate action when the user presses DEL. Figure 8-19 shows a test of `int_flag` after a PROMPT statement. If `int_flag` has been set, this routine presents a message and returns control to the main program. It also resets the `int_flag` variable to 0 so that later tests of `int_flag` will be valid.

```
MAIN
DEFER INTERRUPT
CALL rd_report()
END MAIN

FUNCTION rd_report()
DEFINE edate DATE
LET int_flag = 0

PROMPT "Enter end date for report (or DEL to cancel): " for edate
IF int_flag <> 0 THEN
    MESSAGE "Report cancelled."
    SLEEP 3
    MESSAGE ""
    LET int_flag = 0
    RETURN
END IF
END FUNCTION
```

Figure 8-19: Testing `int_flag` After a PROMPT Statement

SUMMARY

This chapter started by introducing the CONSTRUCT and PREPARE statements and demonstrated how you can set up a query by

example so the user can perform ad hoc queries of the database. It went on to explain more about the use of arrays and detailed the use of the powerful `DISPLAY ARRAY` and `INPUT ARRAY` statements. These statements give you a lot of flexibility in presenting the user with data, both for reviewing and updating.

With the concepts from the whole book and the statements introduced in this chapter, you should be well on your way to building applications using a 4GL.

Appendix A

LISTING OF THE leads PROGRAM

This appendix contains complete listings for all the functions, forms, and reports that make up the **leads** application program. In addition, it shows the simplified versions of two of the functions that are used in Chapter 1. The organization of the appendix is shown in the following table. (Files containing forms end in **.per**, and program files end in **.4gl**.)

PROGRAMS

create.4gl
menu.4gl

FUNCTIONS

contact.4gl
decide.4gl
globals.4gl
product.4gl
prospect.4gl
qcontact.4gl
sperson.4gl
wwait.4gl

FORMS

f_contact.per
f_decide1.per
f_decide2.per
f_decide3.per
f_empnum.per
f_listprod.per
f_pdesc.per
f_prospect.per
f_qcontact.per

FORMS(Continued)

f_sale.per
f_sperson.per

REPORT DRIVERS

rd_follow.4gl
rd_label.4gl
rd_letter.4gl
rd_sales.4gl
rd_sperson.4gl

REPORTS

r_follow.4gl
r_label.4gl
r_letter.4gl
r_sales.4gl
r_sperson.4gl

SIMPLIFIED PROGRAMS

AND FORMS

f_scontact.per
f_sproduct.per
s_contact.4gl
s_medium.4gl
s_product.4gl

PROGRAMS

create.4gl

```

MAIN
{
The create program creates the database tables that the leads
program uses.  It also documents their structure.
}
CREATE DATABASE leads

CREATE TABLE contact
{
Each row in this table summarizes a contact with a prospect.
}
(
  cdate          DATE,
  empnum         SMALLINT,  {Can join with sperson.empnum}
  ndate          DATE,
  nemp          SMALLINT,  {Can join with sperson.empnum}
  notes         CHAR (50),
  ctype         CHAR (1),   {C = complaint
                             I = request for information
                             Q = inquiry
                             S = sale}
  ref            INTEGER,   {Can join with prospect.ref}
  cnum          SERIAL      {Can join with sale.cnum}
)
CREATE INDEX i_cref ON contact(ref)
CREATE UNIQUE INDEX i_cnum ON contact(cnum)

CREATE TABLE prospect
{
This table contains information on the prospect
(both the person and the company).
}
(
  fname          CHAR (15),
  lname          CHAR (15),
  title          CHAR (6),   {letter title: Mr., Ms., Dr., etc.}
  company        CHAR (40),
  add1           CHAR (40),
  add2           CHAR (40),
  add3           CHAR (40),
  city           CHAR (40),
  state          CHAR (2),
  zip            CHAR (5),
  phone          CHAR (12),
  source         CHAR (1),   {B = binder weekly
                             N = newspaper
                             O = other
                             P = binder production news
                             R = radio}
  ref            SERIAL      {This column is for the use of the program;
                             the user never sees it.
                             Can join with contact.ref}
)

```



```
)
CREATE UNIQUE INDEX i_pref ON prospect(ref)

CREATE TABLE sale
{
Each contact that generates a sale references a row in this table
}
(
  cnum          INTEGER,      {Can join with contact.cnum}
  pcode         CHAR (10),    {Can join with product.pcode}
  quantity      SMALLINT,
  discount      SMALLINT      {stored as an integer 1-99}
)

CREATE TABLE sperson
{
Information on each salesperson is kept in this table. The address and
phone are that of the salesperson's business location.
}
(
  empnum        SMALLINT,     {Can join with contact.empnum or
                               Can join with contact.nemp}
  position      CHAR (25),
  lname         CHAR (15),
  fname         CHAR (15),
  add1          CHAR (40),
  add2          CHAR (40),
  add3          CHAR (40),
  city          CHAR (40),
  state         CHAR (2),
  zip           CHAR (5),
  phone         CHAR (12)
)
CREATE UNIQUE INDEX i_sempnum ON sperson(empnum)

CREATE TABLE product
(
  pcode         CHAR (10),    {Can join with sale.pcode}
  price         MONEY(6),
  descrip       CHAR (30)
)
CREATE UNIQUE INDEX i_ppcode ON product(pcode)

GRANT CONNECT TO PUBLIC
END MAIN
```

APPENDIX A. LISTING OF THE leads PROGRAM

menu.4gl

DATABASE leads

GLOBALS "globals.4gl"

MAIN

```
{
The menu program initially displays the Top-Level menu.
Depending on the user's selection, it then either displays a
submenu or calls a function or report driver. When a submenu
is called, it in turn invokes a submenu or calls a function or
report driver, and so forth.
}
```

MENU "TOP-LEVEL"

COMMAND "Contact" "Log information about a contact with a prospect."

CALL decide(1)

IF pr_prospect.ref IS NULL THEN CALL prospect(1) END IF

IF eflag = 0 THEN CALL contact() END IF

LET eflag = 0

COMMAND "Report" "Run a report."

MENU "REPORT"

COMMAND "Mailing-labels" "Generate labels for a prospects mailing."

CALL rd_label()

EXIT MENU

COMMAND "Sales" "Run a report on sales by salesperson by month."

CALL rd_sales()

EXIT MENU

COMMAND "Follow-up" "Run a report on follow-up calls to be made."

CALL rd_follow()

EXIT MENU

COMMAND "Letters" "Generate follow-up letters for contacts."

CALL rd_letter()

EXIT MENU

COMMAND "Exit" "Return to the Top-Level menu."

EXIT MENU

END MENU

COMMAND "Update" "Update information about a contact or prospect."

MENU "UPDATE"

COMMAND "Contact" "Correct information about a contact."

CALL qcontact()

EXIT MENU

COMMAND "Prospect" "Update information about a prospect."

CALL decide(2)

IF pr_prospect.ref IS NOT NULL THEN CALL prospect(2) END IF

EXIT MENU

COMMAND "Exit" "Return to the Top-Level menu."

EXIT MENU

END MENU

COMMAND "Administration" "Change information about a product or salesperson."

MENU "ADMINISTRATION"

COMMAND "Product" "Change information about a product."

CALL product()

```
EXIT MENU
COMMAND "Salesperson" "Change information about a salesperson."
MENU "SALESPERSON"
    COMMAND "Add" "Add a new salesperson."
        CALL sperson(1)
        EXIT MENU
    COMMAND "Update" "Update information about a salesperson."
        CALL sperson(2)
        EXIT MENU
    COMMAND "Delete" "Delete an old salesperson."
        CALL sperson(3)
        EXIT MENU
    COMMAND "List" "Display a list of salespeople."
        CALL rd_sperson()
        EXIT MENU
    COMMAND "Exit" "Return to the Top-Level menu."
        EXIT MENU
END MENU
EXIT MENU
COMMAND "Exit" "Return to the Top-Level menu."
EXIT MENU
END MENU
COMMAND "Exit" "Return to the operating system."
EXIT MENU
END MENU
CLEAR SCREEN
END MAIN
```

FUNCTIONS

contact.4gl

```
DATABASE leads
GLOBALS "globals.4gl"
```

```
FUNCTION contact()
```

```
{
The contact function uses the f_contact form to collect
information from the user. It opens a series of windows
with forms to allow the user to make selections and enter
sales data.
}
```

```
DEFINE      pa_empnum          ARRAY[25] OF LIKE sperson.empnum,
            pa_spers           ARRAY[25] OF CHAR(31),
            pa_pcode           ARRAY[25] OF LIKE product.pcode,
            pa_desc            ARRAY[25] OF LIKE product.descrip,
            pcount, scount, cnt SMALLINT,
            buf1, buf2         CHAR(15),
            msg1               CHAR(59),
            msg2               CHAR(52),
            msg3               CHAR(53),
            msg4               CHAR(39)
```

```
LET msg1 = "Use ARROW keys to move highlight, ESCAPE to make selection."
LET msg2 = "Enter 0 to display and choose from a list of values."
LET msg3 = "Enter C(complaint) I(request info) Q(inquiry) S(sale)"
LET msg4 = "Enter discount as a whole number 0 - 99"
```

```
DECLARE c_sperson CURSOR FOR
SELECT      fname,
            lname,
            empnum
FROM        sperson
ORDER BY    lname,
            fname
```

```
LET scount = 1
FOREACH c_sperson
  INTO      buf1,
            buf2,
            pa_empnum[scount]
  LET pa_spers[scount] = buf1 CLIPPED, 1 SPACE, buf2
  LET scount = scount + 1
END FOREACH
```

```
DECLARE c_prod CURSOR FOR
SELECT      descrip,
            pcode
FROM        product
ORDER BY    pcode
```

```
LET pcount = 1
FOREACH c_prod
  INTO      pa_desc[pcount],
```



```

        pa_pcode[pcount]
    LET pcount = pcount + 1
END FOREACH

INITIALIZE pr_contact.* TO NULL
LET pr_contact.cnum = 0
LET pr_contact.ref = pr_prospect.ref
OPEN FORM f_contact FROM "f_contact"
DISPLAY FORM f_contact

INPUT BY NAME pr_contact.cdate THRU pr_contact.ctype

AFTER FIELD cdate
    LET pr_contact.ndate = pr_contact.cdate + 14
    DISPLAY BY NAME pr_contact.ndate

BEFORE FIELD empnum
    MESSAGE msg2 ATTRIBUTE(REVERSE)

AFTER FIELD empnum
    LET eflag = -1
    IF (pr_contact.empnum IS NOT NULL
        AND pr_contact.empnum != 0) THEN
        SELECT      COUNT(*)
          INTO      cnt
         FROM      sperson
        WHERE      empnum = pr_contact.empnum
        IF (cnt != 1) THEN
            ERROR "There is no salesperson number ",
                pr_contact.empnum USING "###"
        ELSE
            LET eflag = 0
        END IF
    END IF
    IF (eflag < 0) THEN
        LET eflag = 0
        CALL SET_COUNT(scount - 1)
        OPEN WINDOW w_empnum AT 6,10 WITH FORM "f_empnum" ATTRIBUTE(BORDER)
        MESSAGE msg1 ATTRIBUTE(REVERSE)
        DISPLAY ARRAY pa_spers TO sr_con1.*
        CLOSE WINDOW w_empnum
        LET cnt = ARR_CURR()
        LET pr_contact.empnum = pa_empnum[cnt]
        DISPLAY BY NAME pr_contact.empnum
    END IF
    LET pr_contact.nemp = pr_contact.empnum
    DISPLAY BY NAME pr_contact.nemp

BEFORE FIELD nemp
    MESSAGE msg2 ATTRIBUTE(REVERSE)

AFTER FIELD nemp
    LET eflag = -1
    IF (pr_contact.nemp IS NOT NULL
        AND pr_contact.nemp != 0) THEN
        SELECT      COUNT(*)
          INTO      cnt
         FROM      sperson

```

APPENDIX A. LISTING OF THE leads PROGRAM

```
        WHERE      empnum = pr_contact.nemp
    IF (cnt != 1) THEN
        ERROR "There is no salesperson number ",
            pr_contact.nemp USING "###"
    ELSE
        LET eflag = 0
    END IF
END IF
IF (eflag < 0) THEN
    LET eflag = 0
    CALL SET_COUNT(scount - 1)
    OPEN WINDOW w_empnum AT 6,10 WITH FORM "f_empnum" ATTRIBUTE(BORDER)
    MESSAGE msg1 ATTRIBUTE(REVERSE)
    DISPLAY ARRAY pa_spers TO sr_con1.*
    CLOSE WINDOW w_empnum
    LET cnt = ARR_CURR()
    LET pr_contact.nemp = pa_empnum[cnt]
    DISPLAY BY NAME pr_contact.nemp
END IF

BEFORE FIELD ctype
    MESSAGE msg3 ATTRIBUTE(REVERSE)

AFTER FIELD ctype
    MESSAGE ""

END INPUT

IF (pr_contact.ctype = "S") THEN
    OPEN WINDOW w_sale AT 6,10
        WITH 9 ROWS, 55 COLUMNS
        ATTRIBUTE(BORDER)
    OPEN FORM f_sale FROM "f_sale"
    DISPLAY FORM f_sale
    INPUT BY NAME pr_sale.pcode THRU pr_sale.discount

    BEFORE FIELD pcode
        MESSAGE msg2 ATTRIBUTE(REVERSE)

    AFTER FIELD pcode
        MESSAGE ""
        LET eflag = -1
        IF (pr_sale.pcode IS NOT NULL
            AND pr_sale.pcode != "0") THEN
            SELECT      COUNT(*)
            INTO        cnt
            FROM        product
            WHERE        pcode = pr_sale.pcode
            IF (cnt != 1) THEN
                ERROR "There is no product code ", pr_sale.pcode USING "###"
            ELSE
                LET eflag = 0
            END IF
        END IF
    END IF
    IF (eflag < 0) THEN
        LET eflag = 0
        CALL SET_COUNT(pcount - 1)
        OPEN WINDOW w_pdesc AT 9,17 WITH FORM "f_pdesc" ATTRIBUTE(BORDER)
```

```

    MESSAGE msg1 ATTRIBUTE(REVERSE)
    DISPLAY ARRAY pa_desc TO sr_con2.*
    CLOSE WINDOW w_pdesc
    LET cnt = ARR_CURR()
    LET pr_sale.pcode = pa_pcode[cnt]
    DISPLAY BY NAME pr_sale.pcode
END IF

BEFORE FIELD discount
    MESSAGE msg4 ATTRIBUTE(REVERSE)

AFTER FIELD discount
    IF (pr_sale.discount < 0 OR pr_sale.discount > 100) THEN
        ERROR "Must be in the range 0-100"
    NEXT FIELD discount
END IF

END INPUT
CLOSE WINDOW w_sale
END IF

INSERT INTO contact VALUES (pr_contact.*)
IF (pr_contact.ctype = "S") THEN
    LET pr_sale.cnum = SQLCA.SQLERRD[2]
    INSERT INTO sale VALUES (pr_sale.*)
END IF

CLEAR SCREEN
END FUNCTION

```

decide.4gl

```

DATABASE leads
GLOBALS "globals.4gl"

FUNCTION decide(uflag)
{
    The decide function displays the f_decide form and
    matches the user's input to rows in the prospect table.
    If an exact match isn't found, the choose function displays
    a list of possible matches for the user to choose from.

    uflag = 1 for query for old or new prospect,
    uflag = 2 for no query (assume old).
}
DEFINE      buf1                CHAR(40),
            new                  CHAR(1),
            uflag, numrows       SMALLINT
INITIALIZE pr_prospect.* TO NULL

DECLARE     c_prospect CURSOR FOR
SELECT      *
            INTO      pr_prospect."

```

APPENDIX A. LISTING OF THE leads PROGRAM

```
FROM      prospect
WHERE     prospect.company = pr_prospect.company
        AND prospect.lname = pr_prospect.lname

DECLARE   c_try1 CURSOR FOR
SELECT    fname,
        lname,
        company,
        ref

FROM      prospect
WHERE     prospect.company = pr_prospect.company
        AND prospect.lname = pr_prospect.lname
ORDER BY  lname,
        fname,
        company

DECLARE   c_try2 CURSOR FOR
SELECT    fname,
        lname,
        company,
        ref

FROM      prospect
WHERE     prospect.company LIKE buf1
        OR prospect.lname = pr_prospect.lname
ORDER BY  lname,
        fname,
        company

IF (uflag = 1) THEN
OPEN FORM f_decide FROM "f_decide1"
DISPLAY FORM f_decide
INPUT BY NAME pr_prospect.lname,
             pr_prospect.company,
             new
ELSE
OPEN FORM f_decide FROM "f_decide2"
DISPLAY FORM f_decide
INPUT BY NAME pr_prospect.lname,
             pr_prospect.company
LET new = "O"
END IF

CLEAR SCREEN
SELECT    COUNT(*)
INTO      numrows
FROM      prospect
WHERE     prospect.company = pr_prospect.company
        AND prospect.lname = pr_prospect.lname

OPEN c_prospect
CASE
    {One row found, not a new prospect.  Use the row.}
WHEN (numrows = 1 AND new = "O")
    FETCH c_prospect

    {More than one row found, not a new prospect.  Review rows.}
WHEN (numrows > 1 AND new = "O")
    CALL choose(uflag, 1, 1)
```



```

{No rows found, not a new prospect. Review possible rows.}
WHEN (numrows = 0 AND new = "O" )
    LET buf1 = pr_prospect.company[1,5], "%"
    CALL choose(uflag, 2, 2)

{All is well, take no action.}
WHEN (numrows = 0 AND new = "N")

{>= 1 row found, it was supposed to be a new prospect. Verify.}
WHEN (numrows >= 1 AND new = "N")
    CALL choose(uflag, 1, 3)

END CASE
CLEAR SCREEN
END FUNCTION

FUNCTION choose(uuflag, uflag, msg)
{
uuflag    is the uflag from the calling routine
uflag     specifies the type of select
msg       specifies the proper message for display
}
DEFINE    pa_ref          ARRAY[25] OF LIKE prospect.ref,
pa_prosp  ARRAY[25] OF RECORD
          fname          CHAR(31),
          company        LIKE prospect.company
END RECORD,
msg1, msg2, msg3, msg4, msg5, msg6, msg7      CHAR(75),
buf1, buf2                                     CHAR(15),
cnt, uuflag, uflag, msg                      SMALLINT

LET msg1 = " There is more than one possible prospect."
LET msg2 =
" The prospect does not exist in the database exactly as you specified it."
LET msg3 = " The prospect may already exist in the database."
LET msg4 =
" Use the ARROW keys to move the cursor, press ESCAPE to make a selection."
LET msg5 = " Select the first entry if the name you want is not on the list."
LET msg6 = " This is a new prospect."
LET msg7 = " Prospect does not exist in the database."
LET pa_prosp[1].fname = ">>> NOT ON THE LIST <<<"
LET pa_prosp[1].company = NULL
LET cnt = 2

IF (uflag = 1) THEN
    FOREACH c_try1
        INTO    buf1,
                buf2,
                pa_prosp[cnt].company,
                pa_ref[cnt]
        LET pa_prosp[cnt].fname = buf1 CLIPPED, 1 SPACE, buf2
        LET cnt = cnt + 1
    END FOREACH
ELSE
    FOREACH c_try2
        INTO    buf1,
                buf2,
                pa_prosp[cnt].company,

```

APPENDIX A. LISTING OF THE leads PROGRAM

```
        pa_ref[cnt]
    LET pa_prosp[cnt].fname = buf1 CLIPPED, 1 SPACE, buf2
    LET cnt = cnt + 1
END FOREACH
END IF
CALL SET_COUNT(cnt - 1)
IF (cnt = 2) THEN
    IF (uuflag = 1) THEN
        DISPLAY msg6 AT 3,1 ATTRIBUTE(REVERSE)
        SLEEP 3
        INITIALIZE pr_prospect.* TO NULL
    ELSE
        DISPLAY msg7 AT 3,1 ATTRIBUTE(REVERSE)
        SLEEP 3
    END IF
END IF
ELSE
    OPEN FORM f_decide FROM "f_decide3"
    DISPLAY FORM f_decide
    IF (msg = 1) THEN DISPLAY msg1 AT 3,1 ATTRIBUTE(REVERSE) END IF
    IF (msg = 2) THEN DISPLAY msg2 AT 3,1 ATTRIBUTE(REVERSE) END IF
    IF (msg = 3) THEN DISPLAY msg3 AT 3,1 ATTRIBUTE(REVERSE) END IF
    DISPLAY msg4 AT 5,1 ATTRIBUTE(REVERSE)
    DISPLAY msg5 AT 6,1 ATTRIBUTE(REVERSE)
    DISPLAY ARRAY pa_prosp to sr_decide.*
    LET cnt = ARR_CURR()
    IF (cnt = 1) THEN
        INITIALIZE pr_prospect.* TO NULL
    ELSE
        SELECT      *
            INTO      pr_prospect.*
            FROM      prospect
            WHERE      prospect.ref = pa_ref[cnt]
    END IF
END IF
END FUNCTION
```

globals.4gl

DATABASE leads

GLOBALS

}

The globals.4gl file contains a list of global variables.

Each function that defines globals.4gl in a GLOBALS statement can reference any of the variables declared in this file.

}

```
DEFINE      pr_prospect      RECORD LIKE prospect.*,
            pr_contact      RECORD LIKE contact.*,
            pr_sale          RECORD LIKE sale.*,
            pr_sperson       RECORD LIKE sperson.*,
            pr_product       RECORD LIKE product.*,
            eflag            SMALLINT
```

END GLOBALS

product.4gl

```

DATABASE leads
GLOBALS "globals.4gl"

FUNCTION product()
{
The product function displays the f_listprod form and,
depending on the user's input, it adds, deletes, or updates
rows in the product table.
}
DEFINE   pa_prod                      ARRAY[20] OF RECORD LIKE product.*,
        idx, iflag, scrn, cnt        SMALLINT

DECLARE c_prod CURSOR FOR
SELECT   *
        INTO      pr_product.*
        FROM      product
        ORDER BY  pcode
LET idx = 0
FOREACH c_prod
    LET idx = idx + 1
    LET pa_prod[idx].* = pr_product.*
END FOREACH
CALL SET_COUNT(idx)

CLEAR SCREEN
OPEN FORM f_listprod FROM "f_listprod"
DISPLAY FORM f_listprod

INPUT ARRAY pa_prod WITHOUT DEFAULTS FROM sr_product.*
BEFORE ROW
    LET idx = ARR_CURR()
    LET scrn = SCR_LINE()
    LET pr_product.* = pa_prod[idx].*
    LET iflag = 0

ON KEY (CONTROL-B)
    LET pa_prod[idx].* = pr_product.*
    DISPLAY pa_prod[idx].* TO sr_product[scrn].*
    NEXT FIELD pcode

AFTER FIELD pcode
    IF (pa_prod[idx].pcode IS NULL) THEN
        IF (pa_prod[idx].price IS NOT NULL
            OR pa_prod[idx].descrip IS NOT NULL) THEN
            ERROR "You must enter a product code."
            NEXT FIELD pcode
        END IF
    ELSE
        IF (pa_prod[idx].pcode != pr_product.pcode
            OR pr_product.pcode IS NULL) THEN
            SELECT      COUNT(*)
            INTO        cnt
            FROM        product
            WHERE        pcode = pa_prod[idx].pcode
            IF (cnt != 0) THEN

```

APPENDIX A. LISTING OF THE leads PROGRAM

```
        ERROR "Product code must be unique."
    NEXT FIELD pcode
END IF
END IF
END IF

AFTER FIELD price
    IF (pa_prod[idx].price IS NULL
    AND (pa_prod[idx].pcode IS NOT NULL
    OR pa_prod[idx].descrip IS NOT NULL)) THEN
        ERROR "You must enter a price."
    NEXT FIELD price
END IF

AFTER FIELD descrip
    IF (pa_prod[idx].descrip IS NULL
    AND (pa_prod[idx].pcode IS NOT NULL
    OR pa_prod[idx].price IS NOT NULL)) THEN
        ERROR "You must enter a description."
    NEXT FIELD descrip
END IF

BEFORE INSERT
    INITIALIZE pr_product.* TO NULL

AFTER INSERT
    LET iflag = -1
    LET eflag = 0
    IF (pa_prod[idx].pcode IS NOT NULL) THEN
        WHENEVER ERROR CONTINUE
        INSERT INTO product VALUES (pa_prod[idx].*)
        IF (status < 0) THEN LET eflag = -1 END IF
        WHENEVER ERROR STOP
    ELSE
        LET eflag = -1
    END IF
    IF (eflag < 0) THEN
        ERROR "An error has occurred. Please enter the information again."
        INITIALIZE pa_prod[idx].* TO NULL
        CLEAR sr_product[scrn].*
        LET eflag = 0
    END IF

AFTER DELETE
    LET iflag = -1
    DELETE FROM product WHERE pcode = pr_product.pcode

AFTER ROW
    {All nulls, set flag to ignore row.}
    IF (pa_prod[idx].pcode IS NULL
    AND pa_prod[idx].price IS NULL
    AND pa_prod[idx].descrip IS NULL) THEN
        LET iflag = -1
    END IF

    {User made a change to the row: update.}
    IF (iflag = 0
    AND (pr_product.pcode != pa_prod[idx].pcode
```

```

        OR pr_product.price != pa_prod[idx].price
        OR pr_product.descrip != pa_prod[idx].descrip)) THEN
UPDATE product SET
    product.* = pa_prod[idx].*
WHERE pcode = pr_product.pcode
END IF

{User entered new data in a previously null row: insert.}
IF (iflag = 0
    AND (pa_prod[idx].pcode IS NOT NULL
        AND pr_product.pcode IS NULL)) THEN
WHENEVER ERROR CONTINUE
INSERT INTO product VALUES (pa_prod[idx].*)
IF (status < 0) THEN
    ERROR "An error has occurred. Please enter the information again."
    INITIALIZE pa_prod[idx].* TO NULL
    CLEAR sr_product[scrn].*
END IF
WHENEVER ERROR STOP
END IF
END INPUT
CLEAR SCREEN
END FUNCTION

```

prospect.4gl

```

DATABASE leads
GLOBALS "globals.4gl"

FUNCTION prospect(uflag)
{
The prospect function displays the f_prospect form.
Depending on the calling argument, it either inserts or
updates a row in the prospect table.

uflag = 1 to add a new prospect,
uflag = 2 to update an existing prospect
}
DEFINE    uflag          SMALLINT

OPEN FORM f_prospect FROM "f_prospect"
DISPLAY FORM f_prospect

```

APPENDIX A. LISTING OF THE leads PROGRAM

```
INPUT BY NAME pr_prospect.fname THRU pr_prospect.source WITHOUT DEFAULTS
IF (uflag = 1) THEN
    LET pr_prospect.ref = 0
    INSERT INTO prospect VALUES (pr_prospect.*)
    LET pr_prospect.ref = SQLCA.SQLERRD[2]
ELSE
    UPDATE prospect
        SET prospect.* = pr_prospect.*
        WHERE ref = pr_prospect.ref
END IF
CLEAR SCREEN
END FUNCTION
```

qcontact.4gl

```
DATABASE leads
GLOBALS "globals.4gl"
```

```
FUNCTION qcontact()
```

```
;
The qcontact function constructs a query by example based on
the user's input in the f_qcontact form. It retrieves contacts
that satisfy the query and displays each one. The contact the user
selects to update is then updated in the contact table, and if
the contact was a sale, in the sale table.
```

```
;
DEFINE      wbuf      CHAR(400),
            qbuf      CHAR(500),
            answer     CHAR(1),
            cnt        SMALLINT
```

```
CLEAR SCREEN
```

```
OPEN FORM f_qcontact FROM "f_qcontact"
```

```
DISPLAY FORM f_qcontact
```

```
CONSTRUCT wbuf ON
```

```
    contact.cdate,
    contact.empnum,
    contact.ndate,
    contact.nemp,
    contact.notes,
    contact.ctype,
    prospect.lname,
    prospect.company,
    sale.pcode,
    sale.quantity,
    sale.discount,
```

```
    FROM sr consale.*
```

```
LET qbuf =
```

```
    "SELECT      * ",
    "FROM        contact, prospect, OUTER sale ",
    "WHERE        sale.cnum = contact.cnum AND ",
    "              prospect.ref = contact.ref AND ",
    wbuf CLIPPED
```

```

PREPARE q_1 FROM qbuf
DECLARE c_qcontact CURSOR FOR q_1
LET eflag = -1

FOREACH c_qcontact INTO pr_contact.*, pr_prospect.*, pr_sale.*
  LET eflag = 1
  DISPLAY BY NAME
    pr_contact.cdate THRU pr_contact.ctype,
    pr_prospect.lname, pr_prospect.company,
    pr_sale.pcode THRU pr_sale.discount
  PROMPT "RETURN for next row; u to update this row."
  FOR CHAR answer
  IF (answer IS NOT NULL) THEN
    LET eflag = 0
    EXIT FOREACH
  END IF
END FOREACH

IF (eflag = 0) THEN
  INPUT BY NAME
    pr_contact.cdate THRU pr_contact.ctype,
    pr_sale.pcode THRU pr_sale.discount
  WITHOUT DEFAULTS

  AFTER FIELD empnum
    SELECT      COUNT(*)
      INTO      cnt
      FROM      sperson
      WHERE      empnum = pr_contact.empnum
    IF (cnt != 1) THEN
      ERROR "There is no salesperson number ",
        pr_contact.empnum USING "###"
      CLEAR empnum
      LET pr_contact.empnum = NULL
    NEXT FIELD empnum
  END IF

  AFTER FIELD nemp
    SELECT      COUNT(*)
      INTO      cnt
      FROM      sperson
      WHERE      empnum = pr_contact.nemp
    IF (cnt != 1) THEN
      ERROR "There is no salesperson number ",
        pr_contact.nemp USING "###"
      CLEAR nemp
      LET pr_contact.nemp = NULL
    NEXT FIELD nemp
  END IF

  AFTER FIELD ctype
    IF (pr_contact.ctype != "S"
      OR pr_contact.ctype IS NULL) THEN
      CLEAR pcode
      CLEAR quantity
      CLEAR discount
      LET pr_sale.pcode = NULL
      LET pr_sale.quantity = NULL
    
```

APPENDIX A. LISTING OF THE leads PROGRAM

```
        LET pr_sale.discount = NULL
        EXIT INPUT
    END IF

    AFTER FIELD pcode
        SELECT      COUNT(*)
        INTO        cnt
        FROM        product
        WHERE       pcode = pr_sale.pcode
    IF (cnt != 1) THEN
        ERROR "There is no product code ", pr_sale.pcode USING "###"
        CLEAR pcode
        LET pr_sale.pcode = NULL
    NEXT FIELD pcode
    END IF

    AFTER FIELD discount
        IF (pr_sale.discount < 0
            OR pr_sale.discount > 100) THEN
            ERROR "Must be in the range 0-100"
        NEXT FIELD discount
    END IF
END INPUT

UPDATE contact SET
    cdate =      pr_contact.cdate,
    empnum =     pr_contact.empnum,
    ndate =      pr_contact.ndate,
    nemp =       pr_contact.nemp,
    notes =      pr_contact.notes,
    ctype =      pr_contact.ctype
WHERE cnum =    pr_contact.cnum

UPDATE sale SET
    pcode =      pr_sale.pcode,
    quantity =   pr_sale.quantity,
    discount =   pr_sale.discount
WHERE cnum =    pr_sale.cnum
END IF
IF (eflag = -1) THEN
    ERROR "There are no rows that satisfy these criteria."
    SLEEP 3
END IF
IF (eflag = 1) THEN
    ERROR "There are no more rows."
    SLEEP 3
END IF
LET eflag = 0
CLEAR SCREEN
END FUNCTION
```

sperson.4gl

```

DATABASE leads
GLOBALS "globals.4gl"

FUNCTION sperson(uflag)
{
  The sperson function either adds, updates, or deletes a
  salesperson from the sperson table depending on the value of
  the calling argument. In each case it uses the f_sperson form.

  uflag =  1 to add a new salesperson
           2 to update information on an existing salesperson
           3 to delete an existing salesperson
}
DEFINE      uflag, cnt      SMALLINT,
            empno          CHAR (10),
            answer          CHAR(1)

OPEN FORM f_sperson FROM "f_sperson"
CLEAR SCREEN

CASE (uflag)
  WHEN 1
    DISPLAY FORM f_sperson
    INPUT BY NAME pr_sperson.*
    AFTER FIELD empnum
      SELECT      COUNT(*)
        INTO      cnt
        FROM      sperson
        WHERE      empnum = pr_sperson.empnum
      IF (cnt != 0) THEN
        ERROR "Number in use."
        NEXT FIELD empnum
      END IF
      IF (pr_sperson.empnum = 0
        OR pr_sperson.empnum IS NULL) THEN
        ERROR "Enter another employee number "
        NEXT FIELD empnum
      END IF
    END INPUT
    INSERT INTO sperson VALUES (pr_sperson.*)

  WHEN 2
    PROMPT "Enter an employee number (or just RETURN): " FOR empno
    IF (empno IS NULL) THEN
      CALL rd_sperson()
      PROMPT "\nEnter an employee number: " FOR empno
    END IF
    SELECT      *
      INTO      pr_sperson.*
      FROM      sperson
      WHERE      empnum = empno
    IF (status = NOTFOUND) THEN
      ERROR "There is no employee number ", empno USING "###"
      SLEEP 3
    EXIT CASE

```

APPENDIX A. LISTING OF THE leads PROGRAM

```
END IF
DISPLAY FORM f_sperson
DISPLAY BY NAME pr_sperson.empnum
INPUT BY NAME pr_sperson.position THRU pr_sperson.phone WITHOUT DEFAULTS
UPDATE sperson
    SET sperson.* = pr_sperson.*
    WHERE empnum = pr_sperson.empnum

WHEN 3
    PROMPT "Enter an employee number (or just press RETURN ",
        "for a list of employees): " FOR empno
    IF (empno IS NULL) THEN
        CALL rd_sperson()
        PROMPT "\nEnter an employee number: " FOR empno
    END IF
    SELECT      *
        INTO    pr_sperson.*
        FROM    sperson
        WHERE   empnum = empno
    IF (status = NOTFOUND) THEN
        ERROR "There is no employee number ", empno USING "###"
        SLEEP 3
        EXIT CASE
    END IF
    DISPLAY FORM f_sperson
    DISPLAY BY NAME pr_sperson.*
    PROMPT "Delete this sales person? (y/n) " FOR CHAR answer
    IF (answer = "y" OR answer = "Y") THEN
        DELETE FROM sperson WHERE empnum = empno
    END IF
END CASE
CLEAR SCREEN
END FUNCTION
```

wwait.4gl

```
FUNCTION wwait()
{
    The wwait function waits for the user to press a key and then it
    returns control to the calling function.
}
DEFINE    dummy                CHAR(1)
PROMPT "\nPress any key to continue." FOR CHAR dummy
END FUNCTION
```

FORMS

f__contact.per

DATABASE leads

SCREEN

{

CONTACT INFORMATION:

	Date	[c01	]
	Salesperson	[c02	]
	Next	[c03	]
Next salesperson		[c04	]
Comments		[c05	]
Type		[c	]

}

TABLES contact

ATTRIBUTES

c01 = contact.cdate, DEFAULT = TODAY, AUTONEXT;
 c02 = contact.empnum, AUTONEXT;
 c03 = contact.ndate, AUTONEXT;
 c04 = contact.nemp, AUTONEXT;
 c05 = contact.notes;
 c = contact.ctype, UPSHIFT, INCLUDE = (C, I, Q, S), AUTONEXT;

INSTRUCTIONS

DELIMITERS " "

f__decide1.per

DATABASE FORMONLY

SCREEN

{

Please enter the following
 information about the prospect:

Last name	[lname	]
Company name	[company	]
Prospect type	[a	]

}

ATTRIBUTES

lname = FORMONLY.lname;

APPENDIX A. LISTING OF THE leads PROGRAM

```
company = FORMONLY.company, UPSHIFT;  
a      = FORMONLY.new TYPE CHAR, UPSHIFT,  
      INCLUDE = ("N", "O"),  
      COMMENTS = "Enter N for new, O for old or don't know.",  
      REQUIRED, AUTONEXT;
```

f__decide2.per

DATABASE FORMONLY

SCREEN
;

Please enter the following
information about the prospect:

```
      Last name [lname  
Company name [company  
;
```

ATTRIBUTES
lname = FORMONLY.lname;
company = FORMONLY.company, UPSHIFT;

f__decide3.per

DATABASE FORMONLY

SCREEN
;

Name of person		Company name	
[name	]	[company	]
[name	]	[company	]
[name	]	[company	]
[name	]	[company	]
[name	]	[company	]
[name	]	[company	]
[name	]	[company	]
[name	]	[company	]

```
[ name ] [ company ]
[ name ] [ company ]
}
```

ATTRIBUTES

```
name      = FORMONLY.fname;
company   = FORMONLY.company;
```

INSTRUCTIONS

```
DELIMITERS  " "
SCREEN RECORD sr_decide[10] (FORMONLY.fname, FORMONLY.company);
```

f__empnum.per

DATABASE FORMONLY

SCREEN

```
{
```

```
  [ c06 ]
  [ c06 ]
  [ c06 ]
  [ c06 ]
  [ c06 ]
}
```

ATTRIBUTES

```
c06      = FORMONLY.fname;
```

INSTRUCTIONS

```
DELIMITERS  " "
SCREEN RECORD sr_con1[5] (FORMONLY.fname);
```

f__listprod.per

DATABASE leads

SCREEN

```
{
```

PRODUCT INFORMATION:

Use ARROW keys and RETURN to move between fields and rows.
Press FUNCTION KEY 1 to insert a new row.
Press FUNCTION KEY 2 to delete a row.
Press CONTROL-B before you leave a row to change it back to the way it was.
Press ESCAPE to exit.

Code	Price	Description
------	-------	-------------

APPENDIX A. LISTING OF THE leads PROGRAM

```
[cod]    [pri]    [des]
[cod]    [pri]    [des]
[cod]    [pri]    [des]
[cod]    [pri]    [des]
[cod]    [pri]    [des]
}
```

TABLES product

ATTRIBUTES
cod = product.pcode, AUTONEXT;
pri = product.price, AUTONEXT;
des = product.descrip;

INSTRUCTIONS
DELIMITERS " "
SCREEN RECORD sr_product[5] (product.pcode THRU product.descrip)

f__pdesc.per

DATABASE FORMONLY
SCREEN

```
{
s04
s04
s04
s04
s04
}
```

ATTRIBUTES
s04 = FORMONLY.descrip;

INSTRUCTIONS
DELIMITERS " "
SCREEN RECORD sr_con2[5] (FORMONLY.descrip);

f__prospect.per

DATABASE leads

SCREEN
}

PROSPECT INFORMATION:

```

First name      [ p01                ]
Last name      [ p02                ]
Title          [ p03          ]
Company name    [ p04                                ]
Address 1      [ p05                                ]
Address 2      [ p06                                ]
Address 3      [ p07                                ]
City           [ p08                                ]
State          [ p9]
Zip            [ p10          ]
Phone number   [ p11                ]
Source of lead [ p]
}

```

TABLES prospect

ATTRIBUTES

```

p01 = prospect.fname, AUTONEXT;
p02 = prospect.lname, AUTONEXT;
p03 = prospect.title, AUTONEXT;
p04 = prospect.company, UPSHIFT, AUTONEXT;
p05 = prospect.add1, AUTONEXT;
p06 = prospect.add2, AUTONEXT;
p07 = prospect.add3, AUTONEXT;
p08 = prospect.city, AUTONEXT;
p9  = prospect.state, UPSHIFT, AUTONEXT;
p10 = prospect.zip, AUTONEXT;
p11 = prospect.phone, PICTURE = "###/###-####", AUTONEXT;
p    = prospect.source, UPSHIFT, INCLUDE = (B, N, O, P, R), COMMENTS =
      "B=Bind. Wkly N=Newspaper O=Other P=Bind. Prod. News R=Radio",
      AUTONEXT;

```

f_qcontact.per

DATABASE leads
SCREEN

{
UPDATE CONTACT INFORMATION

To display the row for updating, fill in some fields, then press ESCAPE.
Once the row is displayed, update the appropriate fields, then press ESCAPE.

CONTACT INFORMATION:

```

      Date [ c01                ]
Salesperson [ c02          ]
      Next [ c03                ]
Next salesperson [ c04          ]
      Comments [ c05                ]
      Type [ c]

```

PROSPECT INFORMATION:

```

      Last name [ p01                ]
Company name [ p02                ]

```

APPENDIX A. LISTING OF THE leads PROGRAM

SALES INFORMATION:

```
    Product code [s01]
    Quantity sold [s02 ]
    Discount [s3]
```

```
}
```

TABLES contact prospect sale

ATTRIBUTES

```
c01  = contact.cdate;
c02  = contact.empnum;
c03  = contact.ndate;
c04  = contact.nemp;
c05  = contact.notes;
c    = contact.ctype, UPSHIFT, INCLUDE = (C, I, Q, S),
      COMMENTS = "S=sale C=complaint I=info Q=query", AUTONEXT;
p01  = prospect.lname;
p02  = prospect.company, UPSHIFT;
s01  = sale.pcode;
s02  = sale.quantity;
s3   = sale.discount;
```

INSTRUCTIONS

```
SCREEN RECORD sr_consale (contact.cdate THRU sale.discount);
```

f__sale.per

DATABASE leads

SCREEN

```
{
```

SALES INFORMATION:

```
    Product code[s01]
    Quantity sold[s02 ]
    Discount[s3]
```

```
}
```

TABLES sale

ATTRIBUTES

```
s01  = sale.pcode, AUTONEXT;
s02  = sale.quantity, AUTONEXT;
s3   = sale.discount, AUTONEXT;
```

INSTRUCTIONS

```
DELIMITERS " "
```

f__sperson.per

DATABASE leads
SCREEN

{

SALESPERSON INFORMATION

Employee number	[s01]		
Position	[s02]		
Last name	[s03]		
First name	[s04]		
Address 1	[s05]		
Address 2	[s06]		
Address 3	[s07]		
City	[s08]		
State	[a9]		
Zip	[s10]		
Phone	[s11]		

TABLES sperson

ATTRIBUTES

```

s01  = sperson.empnum, AUTONEXT;
s02  = sperson.position, AUTONEXT;
s03  = sperson.lname, AUTONEXT;
s04  = sperson.fname, AUTONEXT;
s05  = sperson.add1, AUTONEXT;
s06  = sperson.add2, AUTONEXT;
s07  = sperson.add3, AUTONEXT;
s08  = sperson.city, AUTONEXT;
a9   = sperson.state, UPSHIFT, AUTONEXT;
s10  = sperson.zip, AUTONEXT;
s11  = sperson.phone, PICTURE = "###/###-####", AUTONEXT;

```

REPORT DRIVERS

rd__follow.4gl

```
DATABASE leads
GLOBALS "globals.4gl"
```

```
FUNCTION rd_follow()
```

```
{
The rd_follow report driver retrieves information about follow-up
contacts for each salesperson for a specified week. It passes the
information to the r_follow report.
}
```

```
DEFINE      pr_follow      RECORD
            cndate          LIKE contact.ndate,
            cref            LIKE contact.ref,
            pfirst          LIKE prospect.fname,
            plast           LIKE prospect.lname,
            pcompany        LIKE prospect.company,
            pphone          LIKE prospect.phone,
            psource         LIKE prospect.source,
            sfirst          LIKE sperson.fname,
            slast           LIKE sperson.lname,
            emp             LIKE sperson.empnum,
            answer          DATE
END RECORD
```

```
CLEAR SCREEN
```

```
PROMPT "Report for week starting on (enter date as mm/dd/yy): "
```

```
FOR pr_follow.answer
```

```
DISPLAY "Running report, output going to follow.out" AT 15,1
```

```
DECLARE c_follow CURSOR FOR
```

```
SELECT      contact.ndate,
            contact.ref,
            prospect.fname,
            prospect.lname,
            prospect.company,
            prospect.phone,
            prospect.source,
            sperson.fname,
            sperson.lname,
            sperson.empnum
```

```
INTO        pr_follow.*
```

```
FROM        contact,
            prospect,
            sperson
```

```
WHERE       contact.ndate
```

```
            BETWEEN pr_follow.answer AND pr_follow.answer + 6
```

```
            AND contact.ref = prospect.ref
```

```
            AND contact.nemp = sperson.empnum
```

```
ORDER BY    sperson.lname,
```

```
            sperson.fname,
```

```
            contact.ndate
```

```
START REPORT r_follow
```



```
FOREACH c_follow
  OUTPUT TO REPORT r_follow(pr_follow.*)
END FOREACH
FINISH REPORT r_follow
DISPLAY "Report finished, output in follow.out", "" AT 15,1
SLEEP 3
DISPLAY "" AT 15,1
END FUNCTION
```

rd_label.4gl

```
DATABASE leads
GLOBALS "globals.4gl"

FUNCTION rd_label()
{
  The rd_label report driver retrieves prospects in zip code order
  from the prospect table. It then passes the information to the
  r_label report.
}
DECLARE c_label CURSOR FOR
  SELECT      *
    INTO      pr_prospect.*
    FROM      prospect
    ORDER BY  zip
DISPLAY "Running report, output going to label.out" AT 15,1
START REPORT r_label
FOREACH c_label
  OUTPUT TO REPORT r_label(pr_prospect.*)
END FOREACH
FINISH REPORT r_label
DISPLAY "Report finished, output in label.out", "" AT 15,1
SLEEP 3
DISPLAY "" AT 15,1
END FUNCTION
```

rd_letter.4gl

```
DATABASE leads
GLOBALS "globals.4gl"

FUNCTION rd_letter()
{
  The rd_letter report driver retrieves contacts for a user-selected
  date and feeds them to the r_letter report.
}
DEFINE      answer      DATE
```

APPENDIX A. LISTING OF THE leads PROGRAM

```
CLEAR SCREEN
PROMPT "Letters for contacts on (enter date as mm/dd/yy or RETURN for today): "
FOR answer
DISPLAY "Running report, output going to letter.out" AT 15,1
IF answer IS NULL THEN LET answer = TODAY END IF
DECLARE c_letter CURSOR FOR
  SELECT      *
    INTO      pr_prospect.*,
              pr_sperson.*,
              pr_contact.*
  FROM        prospect,
              sperson,
              contact
  WHERE       prospect.ref = contact.ref
              AND contact.empnum = sperson.empnum
              AND contact.cdate = answer

LET eflag = -1
START REPORT r_letter
FOREACH c_letter
  LET eflag = 0
  OUTPUT TO REPORT r_letter(pr_prospect.*, pr_sperson.*, pr_contact.*)
END FOREACH
IF (eflag < 0) THEN
  ERROR "There were no contacts on ", answer
  LET eflag = 0
ELSE
  DISPLAY "Report finished, output in letter.out", "" AT 15,1
END IF
FINISH REPORT r_letter
SLEEP 3
DISPLAY "" AT 15,1
END FUNCTION
```

rd_sales.4gl

```
DATABASE leads
GLOBALS "globals.4gl"
```

```
FUNCTION rd_sales()
```

```
{
The rd_sales report driver extracts data for sales occurring
within a six month period and computes the value of sales for
each salesperson for each month. It then feeds the values to
the r_sales report.
}
```

```
DEFINE      msales                MONEY,
           sdate, edate           DATE,
           temp1, smonth, emp, mnth SMALLINT,
           dum1, dum2             CHAR(15)
```

```
CLEAR SCREEN
PROMPT "Enter end date for report as mm/dd/yy (or just RETURN for today): "
      for edate
```

```
DISPLAY "Running report, please wait." AT 15,1

IF edate IS NULL THEN LET edate = TODAY END IF
LET temp1 = MONTH(edate) - 5
IF (temp1 <= 0) THEN
    LET sdate = MDY(12 + temp1, "1", YEAR(edate) - 1)
ELSE
    IF (temp1 = 7) THEN {correct for wrong year: report run in Jan, end in Dec}
        LET sdate = MDY(temp1, "1", YEAR(edate) - 1)
    ELSE
        LET sdate = MDY(temp1, "1", YEAR(edate))
    END IF
END IF
LET smonth = MONTH(sdate)

SELECT      sale.quantity * product.price * (1 - sale.discount/100) tot,
            MONTH(contact.cdate) mon,
            contact.empnum num
FROM        sale,
            contact,
            product
WHERE       sale.cnum = contact.cnum
            AND sale.pcode = product.pcode
            AND contact.cdate BETWEEN sdate AND edate
INTO TEMP hold

DECLARE c_sale CURSOR FOR
SELECT      mon,
            num,
            SUM (tot),
            lname,
            fname
INTO        mnth,
            emp,
            msales,
            dum1,
            dum2
FROM        hold,
            sperson
WHERE       empnum = num
GROUP BY   lname,
            fname,
            num,
            mon
ORDER BY   lname,
            fname

START REPORT r_sales
FOREACH c_sale
    OUTPUT TO REPORT r_sales(sdate, edate, mnth, msales, emp)
END FOREACH
FINISH REPORT r_sales
DROP TABLE hold
END FUNCTION
```

APPENDIX A. LISTING OF THE leads PROGRAM

rd__sperson.4gl

```
DATABASE leads
GLOBALS "globals.4gl"

FUNCTION rd_sperson()
{
  The rd_sperson report driver retrieves the names and employee
  numbers of salespeople from the sperson table. It then passes
  the data to the r_sperson report.
}
DECLARE c_sperson CURSOR FOR
  SELECT      empnum,
              lname,
              fname
  INTO        pr_sperson.empnum,
              pr_sperson.lname,
              pr_sperson.fname
  FROM        sperson
  ORDER BY   lname,
              fname

CLEAR SCREEN
START REPORT r_sperson
FOREACH c_sperson
  OUTPUT TO REPORT r_sperson(pr_sperson.empnum, pr_sperson.lname,
    pr_sperson.fname)
END FOREACH
FINISH REPORT r_sperson
END FUNCTION
```

REPORTS

r__follow.4gl

DATABASE leads
 GLOBALS "globals.4gl"

REPORT r_follow (rr)

The r_follow report prints follow-up contacts scheduled for each salesperson in a specified week. The report is sent to a file called follow.out.

```

DEFINE      rr              RECORD
           ndate            LIKE contact.ndate,
           ref              LIKE contact.ref,
           pfirst           LIKE prospect.fname,
           plast            LIKE prospect.lname,
           company          LIKE prospect.company,
           phone            LIKE prospect.phone,
           source           LIKE prospect.source,
           sfirst           LIKE sperson.fname,
           slast            LIKE sperson.lname,
           empnum           LIKE sperson.empnum,
           answer           DATE
END RECORD,
           cdate            LIKE contact.cdate,
           notes           LIKE contact.notes,
           ctype           LIKE contact.ctype,
           tfirst          LIKE sperson.fname,
           tlast           LIKE sperson.lname

```

OUTPUT
 REPORT TO "follow.out"

FORMAT

FIRST PAGE HEADER

DECLARE c_prev CURSOR FOR

```

SELECT      contact.cdate,
            contact.notes,
            contact.ctype,
            sperson.lname,
            sperson.fname

```

```

INTO        cdate,
            notes,
            ctype,
            tlast,
            tfirst

```

```

FROM        contact,
            sperson

```

```

WHERE       contact.ref = rr.ref
            AND contact.cdate < rr.answer
            AND contact.empnum = sperson.empnum

```

ORDER BY contact.cdate DESC

PRINT "Contacts for ", rr.sfirst CLIPPED, 1 SPACE, rr.slast

APPENDIX A. LISTING OF THE leads PROGRAM

```
PRINT "For the week of ", rr.answer

PAGE HEADER
PRINT "Contacts for ", rr.sfirst CLIPPED, 1 SPACE, rr.slast
PRINT "For the week of ", rr.answer

BEFORE GROUP OF rr.empnum
SKIP TO TOP OF PAGE

ON EVERY ROW
SKIP 1 LINE
PRINT rr.ndate USING "MMM. DD:"
PRINT rr.pfirst CLIPPED, 1 SPACE, rr.plast
PRINT rr.company CLIPPED, ", Phone: ", rr.phone, ", Source: ";
CASE (rr.source)
  WHEN "B" PRINT "Binder Weekly"
  WHEN "N" PRINT "Newspaper"
  WHEN "O" PRINT "Other"
  WHEN "P" PRINT "Binder Prod News"
  WHEN "R" PRINT "Radio"
END CASE
PRINT "Previous contacts:"
SKIP 1 LINE
LET eflag = -1
FOREACH c_prev
  LET eflag = 0
  PRINT "> ", cdate, " by ", tfirst CLIPPED, 1 SPACE,
    tlast CLIPPED, " Type: ";
  IF ctype = "C" THEN PRINT "Complaint" END IF
  IF ctype = "I" THEN PRINT "Request info" END IF
  IF ctype = "Q" THEN PRINT "Inquiry" END IF
  IF ctype = "S" THEN PRINT "Sale" END IF
  IF (notes IS NOT NULL) THEN
    PRINT " Notes: ", notes
  END IF
END FOREACH
IF (eflag < 0) THEN
  PRINT "(first contact)"
  LET eflag = 0
END IF
SKIP 1 LINE

END REPORT
```

r_label.4gl

```
DATABASE leads
REPORT r_label(rr)
```

```
{
The r_label report prints mailing labels for prospects.
The output is sent to a file called label.out.
}
```

```

DEFINE      rr                      RECORD LIKE prospect.*

OUTPUT
  REPORT TO "label.out"
  LEFT MARGIN 0
  TOP MARGIN 0
  BOTTOM MARGIN 0
  PAGE LENGTH 8 {change to the number of lines between the
                  top of one label and the top of the next}

FORMAT
ON EVERY ROW
  PRINT rr.title CLIPPED, 1 SPACE,
        rr.fname CLIPPED, 1 SPACE,
        rr.lname
  IF rr.company IS NOT NULL THEN PRINT rr.company END IF
  IF rr.add1 IS NOT NULL THEN PRINT rr.add1 END IF
  IF rr.add2 IS NOT NULL THEN PRINT rr.add2 END IF
  IF rr.add3 IS NOT NULL THEN PRINT rr.add3 END IF
  PRINT rr.city CLIPPED, ", ", rr.state, 2 SPACES, rr.zip
  SKIP TO TOP OF PAGE
END REPORT

```

r__letter.4gl

```

DATABASE leads
GLOBALS "globals.4gl"

```

```

REPORT r__letter(rr_p, rr_s, rr_c)

```

```

{
The r__letter report prints letters to prospects, sending
the output to letter.out. The content of each letter differs
depending on whether the contact was a complaint, request for
information, inquiry, or sale.
}

```

```

DEFINE      rr_p                      RECORD LIKE prospect.*,
        rr_s                      RECORD LIKE sperson.*,
        rr_c                      RECORD LIKE contact.*,
        dummy                     CHAR(20)

```

```

OUTPUT
  REPORT TO "letter.out"

```

```

FORMAT
ON EVERY ROW
  SKIP TO TOP OF PAGE
  SKIP 10 LINES
  PRINT rr_p.title CLIPPED, 1 SPACE,
        rr_p.fname CLIPPED, 1 SPACE,
        rr_p.lname
  PRINT rr_p.company
  IF (rr_p.add1 IS NOT NULL) THEN PRINT rr_p.add1 END IF
  IF (rr_p.add2 IS NOT NULL) THEN PRINT rr_p.add2 END IF
  IF (rr_p.add3 IS NOT NULL) THEN PRINT rr_p.add3 END IF

```

APPENDIX A. LISTING OF THE leads PROGRAM

```
PRINT rr_p.city CLIPPED, " ",
      rr_p.state CLIPPED, 2 SPACES,
      rr_p.zip
SKIP 1 LINE
LET dummy = MONTH(rr_c.cdate)
IF dummy = 1 THEN PRINT "January"; END IF
IF dummy = 2 THEN PRINT "February"; END IF
IF dummy = 3 THEN PRINT "March"; END IF
IF dummy = 4 THEN PRINT "April"; END IF
IF dummy = 5 THEN PRINT "May"; END IF
IF dummy = 6 THEN PRINT "June"; END IF
IF dummy = 7 THEN PRINT "July"; END IF
IF dummy = 8 THEN PRINT "August"; END IF
IF dummy = 9 THEN PRINT "September"; END IF
IF dummy = 10 THEN PRINT "October"; END IF
IF dummy = 11 THEN PRINT "November"; END IF
IF dummy = 12 THEN PRINT "December"; END IF
PRINT rr_c.cdate USING " DD, YYYY"
SKIP 1 LINE
PRINT "Dear ",
      rr_p.title CLIPPED, 1 SPACE,
      rr_p.lname CLIPPED, ": "
SKIP 1 LINE
IF (rr_c ctype = "C") THEN
  PRINT "I was sorry to hear that you were not satisfied"
  PRINT "with your recent shipment; if binders I hope the"
  PRINT "issue has been settled to your satisfaction. If"
  PRINT "not, do not hesitate to contact me"
END IF

IF (rr_c ctype = "I") THEN
  PRINT "Thank you for your request for information. I"
  PRINT "will be sending you literature under separate"
  PRINT "cover."
END IF

IF (rr_c ctype = "Q") THEN
  PRINT "Thank you for your recent inquiry. I hope you"
  PRINT "have had all your questions answered. I will be"
  PRINT "sending you our literature under separate cover."
END IF

IF (rr_c ctype = "S") THEN
  PRINT "Thank you for your recent purchase. I hope the"
  PRINT "binders meet your requirements and that we can"
  PRINT "continue to do business together in the future."
END IF

PRINT FILE "para"
IF rr_s.add1 IS NOT NULL THEN PRINT rr_s.add1 END IF
IF rr_s.add2 IS NOT NULL THEN PRINT rr_s.add2 END IF
IF rr_s.add3 IS NOT NULL THEN PRINT rr_s.add3 END IF
PRINT rr_s.city CLIPPED, " ",
      rr_s.state CLIPPED, 2 SPACES,
      rr_s.zip
PRINT "telephone: ", rr_s.phone
SKIP 1 LINE
PRINT "Sincerely,"
```

```

SKIP 4 LINES
PRINT rr_s.fname CLIPPED, 1 SPACE,
      rr_s.lname CLIPPED, " ",
      rr_s.position
PRINT "Associated Binder Company"
END REPORT

```

r_sales.4gl

```
REPORT r_sales(rbegin, rend, mon, msales, emp)
```

```
{
The r_sales report prints the value of sales for a six-month
period, by salesperson and month, with salesperson and month
totals and a grand total. The information is printed in
tabular format on the screen.
}
```

```

DEFINE      msales          INTEGER,
            sfirst          CHAR(1),
            slast           CHAR(15),
            col, totcol     ARRAY[6] OF INTEGER,
            months          ARRAY[12] OF CHAR(3),
            rbegin, rend    DATE,
            mon, emp, temp1 SMALLINT

```

```
OUTPUT
```

```

TOP MARGIN 0
BOTTOM MARGIN 0
LEFT MARGIN 0
PAGE LENGTH 22

```

```
FORMAT
```

```
PAGE HEADER
```

```

IF (PAGENO = 1) THEN
  LET months[1] = "Jan"  LET months[2] = "Feb"
  LET months[3] = "Mar"  LET months[4] = "Apr"
  LET months[5] = "May"  LET months[6] = "Jun"
  LET months[7] = "Jul"  LET months[8] = "Aug"
  LET months[9] = "Sep"  LET months[10] = "Oct"
  LET months[11] = "Nov" LET months[12] = "Dec"
  FOR temp1 = 1 TO 6
    LET totcol[temp1] = 0
  END FOR
END IF

```

```
CLEAR SCREEN
```

```

PRINT COLUMN 35, "S A L E S"
PRINT COLUMN 20, "for the period ",
  rbegin USING "MMM. DD, YYYY", " - ", rend USING "MMM. DD, YYYY"
PRINT COLUMN 33, "by salesperson"
SKIP 1 LINE
LET temp1 = MONTH(rbegin)
PRINT "Salesperson", COLUMN 22, months[temp1];

```

APPENDIX A. LISTING OF THE leads PROGRAM

```
LET temp1 = temp1 + 1
IF temp1 > 12 THEN LET temp1 = temp1 - 12 END IF
PRINT COLUMN 30, months[temp1];
LET temp1 = temp1 + 1
IF temp1 > 12 THEN LET temp1 = temp1 - 12 END IF
PRINT COLUMN 38, months[temp1];
LET temp1 = temp1 + 1
IF temp1 > 12 THEN LET temp1 = temp1 - 12 END IF
PRINT COLUMN 46, months[temp1];
LET temp1 = temp1 + 1
IF temp1 > 12 THEN LET temp1 = temp1 - 12 END IF
PRINT COLUMN 54, months[temp1];
LET temp1 = temp1 + 1
IF (temp1 > 12) THEN LET temp1 = temp1 - 12 END IF
PRINT COLUMN 62, months[temp1], COLUMN 74, "Total"
SKIP 1 LINE
```

ON EVERY ROW

```
IF (MONTH(rbegin) > mon) THEN
    LET mon = mon + 12
END IF
LET mon = mon - MONTH(rbegin) + 1
LET col[mon] = msales
```

BEFORE GROUP OF emp

```
FOR temp1 = 1 TO 6
    LET col[temp1] = 0
END FOR
```

AFTER GROUP OF emp

```
SELECT      fname[1],
            lname
    INTO      sfirst,
            slast
    FROM      sperson
    WHERE     empnum = emp
```

```
PRINT sfirst, 1 SPACE, slast,
    COLUMN 18, col[1] USING "###,##&",
    COLUMN 26, col[2] USING "###,##&",
    COLUMN 34, col[3] USING "###,##&",
    COLUMN 42, col[4] USING "###,##&",
    COLUMN 50, col[5] USING "###,##&",
    COLUMN 58, col[6] USING "###,##&",
    COLUMN 70, col[1] + col[2] + col[3] + col[4] + col[5] + col[6]
                USING "#,###,##&"
```

```
FOR temp1 = 1 TO 6
    LET totcol[temp1] = totcol[temp1] + col[temp1]
END FOR
```

PAGE TRAILER

```
CALL wwait()
```

ON LAST ROW

```
SKIP 1 LINE
PRINT "Total",
    COLUMN 18, totcol[1] USING "###,##&",
    COLUMN 26, totcol[2] USING "###,##&",
```



```
COLUMN 34, totcol[3] USING "###,##&",
COLUMN 42, totcol[4] USING "###,##&",
COLUMN 50, totcol[5] USING "###,##&",
COLUMN 58, totcol[6] USING "###,##&",
COLUMN 70, totcol[1] + totcol[2] + totcol[3]
      + totcol[4] + totcol[5] + totcol[6]
      USING "#,###,##&"
```

END REPORT

r_sperson.4gl

DATABASE leads
GLOBALS "globals.4gl"

REPORT r_sperson(empnum, lnm, fnm)

{
The r_sperson report lists the employee codes and names of salespeople
on the screen.
}

DEFINE empnum LIKE sperson.empnum,
 lnm LIKE sperson.lname,
 fnm LIKE sperson.fname

OUTPUT

TOP MARGIN 0
BOTTOM MARGIN 0
PAGE LENGTH 22

FORMAT

PAGE HEADER
CLEAR SCREEN
PRINT "LIST OF CODES AND SALESPEOPLE"
SKIP 1 LINE
PRINT "Code", COLUMN 15, "Name"
SKIP 1 LINE

PAGE TRAILER
CALL wwait()

ON EVERY ROW
PRINT empnum, COLUMN 15, lnm CLIPPED, ", ", fnm CLIPPED

END REPORT

SIMPLIFIED PROGRAMS AND FORMS

f__scontact.per

```

DATABASE leads
SCREEN
{
    CONTACT INFORMATION:

        Date[ c01          ]
        Salesperson[ c02    ]
        Next       [ c03          ]
        Next salesperson[ c04    ]
        Comments   [ c05          ]
        Type       [ c          ]

}

TABLES contact

ATTRIBUTES
c01  = contact.cdate, DEFAULT = TODAY, AUTONEXT;
c02  = contact.empnum, AUTONEXT;
c03  = contact.ndate, AUTONEXT;
c04  = contact.nemp, AUTONEXT;
c05  = contact.notes;
c    = contact.ctype, UPSHIFT, INCLUDE = (C, I, Q, S), AUTONEXT;

INSTRUCTIONS
DELIMITERS    "    "
```

f__sproduct.per

```

DATABASE leads
SCREEN
{
    PRODUCT INFORMATION (no error checking version):

    Use ARROWS and RETURN to move between fields and rows, ESCAPE to exit.
    Press FUNCTION KEY 1 to insert a new row, FUNCTION KEY 2 to delete a row.

    Code      Price      Description

    [cod]      [pri]      [des
    [cod]      [pri]      [des
    [cod]      [pri]      [des
    [cod]      [pri]      [des
    [cod]      [pri]      [des
}
```

TABLES product

ATTRIBUTES

```
cod  = product.pcode, AUTONEXT;
pri  = product.price, AUTONEXT;
des  = product.descrip;
```

INSTRUCTIONS

```
DELIMITERS  "  "
SCREEN RECORD sr_product[5] (product.pcode THRU product.descrip)
```

s__contact.4gl

DATABASE leads

GLOBALS "globals.4gl"

MAIN

```
{
The s_contact program (a simplified version of the contact
function) displays the f_scontact form to collect information
from the user. It opens a series of windows with forms to
allow the user to enter sales data and select a product code.
}
```

```
DEFINE    pa_pcode      ARRAY[25] OF LIKE product.pcode,
           pa_prod      ARRAY[25] OF LIKE product.descrip,
           cnt, pcount   SMALLINT
```

```
DECLARE c_prod CURSOR FOR
SELECT      descrip,
           pcode
FROM        product
ORDER BY    pcode
```

```
LET pcount = 1
FOREACH c_prod                                {Fill pa_prod array.}
  INTO    pa_prod[pcount],
          pa_pcode[pcount]
  LET pcount = pcount + 1
END FOREACH
```

```
INITIALIZE pr_contact.* TO NULL
LET pr_contact.cnum = 0
LET pr_contact.ref = 1 {Set to 1, no-error version only.}
OPEN FORM f_scontact FROM "f_scontact"
DISPLAY FORM f_scontact
```

```
INPUT BY NAME pr_contact.cdate THRU pr_contact.ctype
```

```
AFTER FIELD cdate
{Set and display next contact 2 weeks from this contact.}
LET pr_contact.ndate = pr_contact.cdate + 14
DISPLAY BY NAME pr_contact.ndate
```

APPENDIX A. LISTING OF THE leads PROGRAM

```
AFTER FIELD empnum
{Set and display next employee to current employee}
  LET pr_contact.nemp = pr_contact.empnum
  DISPLAY BY NAME pr_contact.nemp

BEFORE FIELD ctype
{Display instructions for user.}
  MESSAGE "Enter C(complaint) I(request info) Q(inquiry) S(sale)"
  ATTRIBUTE(REVERSE)

AFTER FIELD ctype
  MESSAGE "" {Clear message line.}

END INPUT

IF (pr_contact.ctype = "S") THEN
  OPEN WINDOW w_sale AT 6,10
  WITH 9 ROWS, 55 COLUMNS
  ATTRIBUTE(BORDER)
  OPEN FORM f_sale FROM "f_sale"
  DISPLAY FORM f_sale

  INPUT BY NAME pr_sale.pcode THRU pr_sale.discount

  BEFORE FIELD pcode
  {Display instructions for user.}
    MESSAGE "Enter 0 to display and choose from a list of values."
    ATTRIBUTE(REVERSE)

  AFTER FIELD pcode
  {list products if user entered 0 or just pressed RETURN}
    MESSAGE "" {Clear message line.}
    IF (pr_sale.pcode IS NULL {See if user just pressed}
      OR pr_sale.pcode = "0") THEN {RETURN or entered 0.}
      CALL SET_COUNT(pcount - 1) {Set up for DISPLAY ARRAY.}
      OPEN WINDOW w_pdesc AT 9,17 WITH FORM "f_pdesc" ATTRIBUTE(BORDER)
      MESSAGE "Use ARROW keys to move highlight, ESCAPE to make selection."
      ATTRIBUTE(REVERSE) {Display message in reverse video.}
      DISPLAY ARRAY pa_prod TO sr_con2.* {Display array on form.}
      CLOSE WINDOW w_pdesc
      LET cnt = ARR_CURR() {Determine which item user picked.}
      LET pr_sale.pcode = pa_pcode[cnt] {record element=array element}
      DISPLAY BY NAME pr_sale.pcode {Show product code on form.}
    END IF

  BEFORE FIELD discount
    MESSAGE "Enter discount as a whole number 0 - 99" ATTRIBUTE(REVERSE)

  AFTER FIELD discount
    MESSAGE ""

  END INPUT
  CLOSE WINDOW w_sale
  END IF

INSERT INTO contact VALUES (pr_contact.*) {Insert row into contact table.}
IF (pr_contact.ctype = "S") THEN {If it was a sale,}
  LET pr_sale.cnum = SQLCA.SQLERRD[2] {get serial number assigned by 4GL}
```

```

      INSERT INTO sale VALUES (pr_sale.*)      {insert row into sale table.}
END IF

END MAIN

```

s__medium.4gl

```

DATABASE leads
GLOBALS "globals.4gl"

```

```

MAIN
{

```

```

The s__medium program computes the value of each sale for each
salesperson in a specified period and feeds them to the
r__medium report. The r__medium report prints the total value
of the sales for each employee as well as the grand total
for the period. The output is sent to medium.out.
}

```

```

DEFINE      pr_ssum          RECORD
            tot              MONEY,
            emp              SMALLINT,
            sdate            DATE,
            edate            DATE
            END RECORD

```

```

DECLARE c_sales CURSOR FOR
SELECT      sale.quantity * product.price * (1 - sale.discount/100),
            contact.empnum
            pr_ssum.tot,
            pr_ssum.emp
FROM        sale,
            product,
            contact
WHERE       sale.cnum = contact.cnum
            AND sale.pcode = product.pcode
            AND contact.cdate BETWEEN pr_ssum.sdate AND pr_ssum.edate
ORDER BY   contact.empnum

```

```

PROMPT "Start Date (mm/dd/yy): " FOR pr_ssum.sdate

```

```

PROMPT "End Date (mm/dd/yy): " FOR pr_ssum.edate

```

```

START REPORT r__medium

```

```

FOREACH c_sales

```

```

    OUTPUT TO REPORT r__medium(pr_ssum.*)

```

```

END FOREACH

```

```

FINISH REPORT r__medium

```

```

END MAIN

```

```

REPORT r__medium(rr)
DEFINE      rr              RECORD
            tot              MONEY,
            emp              SMALLINT,
            sdate            DATE,
            edate            DATE

```


APPENDIX A. LISTING OF THE leads PROGRAM

```
END RECORD

OUTPUT
  REPORT TO "medium.out"

FORMAT
PAGE HEADER
  PRINT "Sales Summary by Salesperson"
  PRINT "For the Period ", rr.sdate, " through ", rr.edate
  SKIP 2 LINES
  PRINT "Employee", COLUMN 20, "Total"
  PRINT "Number", COLUMN 20, "Sales"
  SKIP 1 LINE

AFTER GROUP OF rr.emp
  PRINT rr.emp, COLUMN 15, GROUP SUM(rr.tot) USING "###,##&.&&"

ON LAST ROW
  SKIP 1 LINE
  PRINT "Total", COLUMN 13, SUM(rr.tot) USING "#,###,##&.&&"
END REPORT
```

s__product.4gl

```
DATABASE leads
GLOBALS "globals.4gl"

MAIN
{
  The s__product program (a simplified version of the product
  function) displays the f__product form and allows the user
  to add or delete products. It contains no error checking
  code.
}
DEFINE   pa_prod          ARRAY[20] OF RECORD LIKE product.*,
         idx              SMALLINT

DECLARE c_prod CURSOR FOR
  SELECT   *
  INTO     pr_product.*
  FROM     product
  ORDER BY pcode

LET idx = 0
FOREACH c_prod
  LET idx = idx + 1
  LET pa_prod[idx].* = pr_product.*
END FOREACH
CALL SET_COUNT(idx)

OPEN FORM f__product FROM "f__product"
DISPLAY FORM f__product
```

```
INPUT ARRAY pa_prod WITHOUT DEFAULTS FROM sr_product.*

AFTER INSERT
  LET idx = ARR_CURR()
  INSERT INTO product VALUES (pa_prod[idx].*)

AFTER DELETE
  DELETE FROM product WHERE pcode = pr_product.pcode

END INPUT
END MAIN
```

Appendix B

ABOUT NULLS

All types of variables, from CHAR to INTEGER to DATE, can take on null values. If a variable is null, it means the variable has not been assigned an explicit value; it has no particular value. Null is not zero, nor is it a blank.

You can assign a null value to a variable using LET.

```
LET x = NULL
```

You can initialize all elements of a record variable to null values using INITIALIZE.

```
INITIALIZE pr__contact.* TO NULL
```

Because a null represents an unknown value, when a null participates in an arithmetic expression, the result of the expression is null. In the following example, when x, which has a null value, is added to 5, the result is null. The program displays the string The value is with nothing (a null value) following it.

```
MAIN
DEFINE x SMALLINT
LET x = NULL
LET x = x + 5
DISPLAY "The value is ", x
END MAIN
```

The time nulls get tricky is when you are using a Boolean operator to test to see if a null variable has a certain relationship to a value. Because a null represents an unknown value, you can

never be sure that it has, or does not have, any specific relationship to any other value, even another null value. Although intuitively you might think that the following program would display the word `true`, it does not display anything.

```
MAIN
DEFINE x SMALLINT
LET x = NULL
IF x != 5 THEN DISPLAY "true" END IF
END MAIN
```

The problem with intuition in this case is that it is obvious to you and me that `x` has been assigned a null value so it cannot be equal to 5 and the test following the `IF` statement must be true. But null represents an unknown value. And you can never be sure that that unknown value is not 5. So you can never know conclusively that `x` is not equal to 5.

The solution to this problem is to use the `IS NULL` clause to make an explicit test to see if `x` has a null value. Refer to the following example.

```
MAIN
DEFINE x SMALLINT
LET x = NULL
IF x != 5 OR x IS NULL THEN DISPLAY "true" END IF
END MAIN
```

A practical application of this concept is demonstrated in the `INPUT` statement in the `product` function of the `leads` program. The `AFTER FIELD` clause that gets executed after the user moves the cursor out of the `pcode` field tests to see whether `pcode` is null. Assuming `pcode` is not null, then what happens next depends on whether the user entered data in the field. If the user entered data, the value of `pa__prod[idx].pcode` is no longer equal to the value of `pr__product.pcode`. Instead of being able to use the statement

```
IF (pa__prod[idx].pcode != pr__product.pcode)
```


to see if the user made an entry in the field, you must also test for the case where the field was originally null (as occurs when the user inserts a new product). Along the lines of the previous discussion, if **pr_product.pcode** is null, you can never be sure it is not equal to **pa_prod[idx].pcode** and the IF statement will return a false value. As shown below, you need to test explicitly for a null value.

```
IF (pa_prod[idx].pcode != pr_product.pcode
OR pr_product.pcode IS NULL) THEN
  SELECT      COUNT(*)
  INTO        cnt
  FROM        product
  WHERE       pcode = pa_prod[idx].pcode
  IF (cnt != 0) THEN
    ERROR "Product code must be unique."
  NEXT FIELD pcode
END IF
END IF
```

If you want to prohibit a user from entering a null value in a form field, you can define the field, in the ATTRIBUTES section of the form, to have the WITHOUT NULL INPUT attribute.

SUMMARY

In general, if an arithmetic operation or a logical comparison does not seem to be working in the way you would expect, see if one of the elements of the operation or comparison could have a null value. If it does, isolate the part of the process that involves the null value and treat it as a special case using the IS NULL or IS NOT NULL clause as appropriate. Frequently it is possible to design an application to have either mandatory data entry in fields or default values so the procedural code does not have to take null values into account.

Appendix C

SETTING UP THE **leads** DATABASE AND COMPILING PROGRAMS

The first part of this appendix explains how to install the database you need to work with the **leads** demonstration program. The appendix continues with an explanation of how to use the **c4gl** command to compile and link an INFORMIX-4GL program from the command line. You can also use the INFORMIX-4GL interactive development environment to create, compile, link, and execute INFORMIX-4GL programs—refer to the *INFORMIX-4GL User Guide* for more information.

SETTING UP THE **leads** DATABASE

If you have **INFORMIX-4GL** installed on your computer, you can use the **buildleads** script to build a copy of the **leads** database so you can run the examples in this book. (You cannot compile the programs that make up the **leads** application using a demonstration copy of **INFORMIX-4GL**.) Because the **buildleads** script puts a lot of files in the working directory, it is a good idea to create a new directory and change to that directory before running it. Give the following commands, substituting a name of your choosing in place of **demoleads**:

```
mkdir demoleads
cd demoleads
buildleads
```

With the final command, your system will start building the database and compiling the programs and forms that you need to run the **leads** application. The **buildleads** script will display periodic messages as it runs. Depending on the speed of your computer, this process can take some time. When it has finished, you will have a complete copy of the **leads** application program and database in the working directory. You can build as many copies of this application as you like (provided you have sufficient disk space), but each must be in its own directory.

With the **leads** application installed, you can run it by giving the following command:

```
leads
```

You should then see the **Top-Level** menu. Because there is only a small amount of data in the **leads** database, some reports may not produce much output unless you respond to the report's prompt with one of the dates listed below.

Report	Date to Specify
Sales	6/30/86
Follow-up	3/15/86
Letters	1/15/86

To allow you maximum flexibility, the **buildleads** script is designed to rebuild a database when you execute it from a directory that already contains a copy of the **leads** database. Taking advantage of this feature, you can run the **leads** program and make as many changes to the test data as you like. When you want to reset the database so it contains the original set of data, you can run **buildleads** again. Be aware that in this situation **buildleads** will destroy any data you have entered into the **leads** database.

Refer to Appendix D for a list of the data that the **leads** database contains when you create it.

COMPILING AN INFORMIX-4GL PROGRAM

This section shows how to compile and link a program that you store in a single operating system file as well as one you store in multiple files. It also covers how to use **form4gl** to compile both default and custom forms.

If your main program, and any functions the main program calls, is in a single operating system file, the following command will compile and link the program. The command assumes the file that holds the program is named **accounts.4gl** and that you want the executable file to be named **accounts**. (The name of the executable file must follow the **-o** on the command line.)

```
c4gl -o accounts accounts.4gl
```

If the INFORMIX-4GL compiler finds any errors, you will see a message telling you how many errors it found and the name of the file that contains the error messages. If the compilation is successful, most compilers will leave you with three files in addition to the source and executable files. The filenames will begin with **accounts** and end with the filename extensions **.ec**, **.c**, and **.o**.

You can delete the `.ec` and `.c` files, but you may want to leave the `.o` file for the reason explained below.

For larger programs, it is sometimes convenient to store functions in more than one file. If the `accounts` program calls additional functions that are stored in a file named `acc_func.4gl`, the following command line will compile and link the necessary files:

```
c4gl -o accounts accounts.4gl acc_func.4gl
```

If you are using a `globals` file, its name must also appear on the command line. You can put as many filenames on the command line as you like.

When you are working with multiple files, you may need to recompile only one of the files. If you make no changes to a source file (a file with a `.4gl` filename extension), you do not need to recompile it. That is where the `.o` files come in. These files contain compiled, but as yet unlinked, object code, and you can use them on the command line in place of the `.4gl` files. Assuming that you are working with `accounts.4gl` and `acc_func.4gl` files, and that the `acc_func.4gl` file has been successfully compiled and is working properly, you can recompile the `accounts.4gl` file and link it with the already compiled version of `acc_func.4gl` (which is stored in `acc_func.o`) with the following command line:

```
c4gl -o accounts accounts.4gl acc_func.o
```

You can use the `-c` option if you want to compile, but not link, a file. Compiling in this manner is useful when you want to see if you have removed all the syntax errors from a section of code. The following command line will compile the `acc_func.4gl` file and generate `acc_func.o`. No attempt will be made to link the file, so it can contain a function without a main program.

```
c4gl -c acc_func.4gl
```

Finally, you can use `c4gl` to link already compiled modules:

```
c4gl -o accounts accounts.o acc_func.o
```

BUILDING A FORM

The **form4gl** compiler takes a form definition file (a **.per** file) and turns it into a file that an **INFORMIX-4GL** program can work with (a **.frm** file). You can use the same compiler to create a default form based on one or more tables. As with the other work you do with **INFORMIX-4GL**, you can also use the interactive development environment to create forms. Refer to the *INFORMIX-4GL User Guide* for more information.

The following command and responses create a default form named **temp** based on the **prospect** table in the **leads** database.

```
$ form4gl -d
```

```
INFORMIX-4GL Form Builder                INFORMIX-SQL 2.1a.00
Copyright (C) Relational Database Systems, Inc., 1984-1986
Software Serial Number RDS#R000000
```

```
Enter desired name for the form: temp
Enter the name of the database used by the form: leads
```

```
You will now enter the names of the database tables which will appear on
the form. You may enter a maximum of 8 table names.
Enter a carriage return alone to terminate the sequence.
```

```
Enter a table name: prospect
Enter a table name:
```

```
The form "temp.per" will now be compiled.
```

```

The form compilation was successful.
```

A default form includes fields for all the columns in the tables you specify, labeling each field with the corresponding column name. The preceding command creates two files: the compiled default form (**temp.frm**) and the default form you can edit and then recompile (**temp.per**).

If you just want to compile an existing form, give the command without an option:

```
form4gl temp
```

This command will produce a single **.frm** file.

FILENAME EXTENSIONS

Below is a list of the filename extensions you may come across while using **INFORMIX-4GL**.

.4gl	the INFORMIX-4GL source file
.ec	the output from the INFORMIX-4GL preprocessor (C code with embedded SQL commands)
.c	the output from the Embedded SQL/C compiler (C code)
.o	the unlinked object files
.per	the source file for a form
.frm	the object file for a form

Appendix D

THE leads DATABASE

This appendix was produced by the following SQL statements run under INFORMIX-SQL. It contains a listing of all the data in the leads database that was used for the examples in this book.

```
select * from contact order by ref, cdate;
select * from product order by pcode;
select * from prospect order by ref;
select * from sale order by cnum;
select * from sperson order by empnum;
```

The contact Table

cdate 01/15/1986
empnum 125
ndate 06/01/1986
nemp 125
notes He won't need quantity til summer
ctype S
ref 1
cnum 1

cdate 06/01/1986
empnum 125
ndate 10/01/1986
nemp 126
notes He doesn't want to be called
ctype S
ref 1
cnum 2

cdate 10/01/1986
empnum 126
ndate 11/15/1986
nemp 126
notes he'll be placing more big orders
ctype S
ref 1
cnum 8

cdate 02/05/1986
empnum 127
ndate 02/19/1986
nemp 126
notes Keep in touch with him
ctype S
ref 2
cnum 3

cdate 02/20/1986
empnum 128
ndate 03/17/1986
nemp 128
notes His binders were falling apart
ctype C
ref 2
cnum 4

cdate 04/28/1986
empnum 125
ndate 05/12/1986
nemp 125
notes He was satisfied with the replacement
ctype S
ref 2

cnum 22

cdate 05/15/1986

empnum 125

ndate 05/29/1986

nemp 125

notes

ctype S

ref 2

cnum 23

cdate 06/03/1986

empnum 127

ndate 06/17/1986

nemp 127

notes I'm just filling in for this order

ctype S

ref 2

cnum 24

cdate 12/17/1985

empnum 127

ndate 01/15/1986

nemp 127

notes

ctype S

ref 3

cnum 5

cdate 01/15/1986

empnum 125

ndate 01/29/1986

nemp 125

notes He wants oil proof binders

ctype S

ref 3

cnum 6

cdate 01/29/1986

empnum 125

ndate 03/15/1986

nemp 125

notes He'll settle for our regular binders

ctype S

ref 3

cnum 9

cdate 03/15/1986

empnum 126

ndate 04/15/1986

nemp 126

notes Call him once a month

ctype S

ref 3

cnum 18

cdate 04/18/1986

empnum 126

Appendix D. The **leads** DATABASE

ndate 05/15/1986
nemp 126
notes
ctype S
ref 3
cnum 19

cdate 05/15/1986
empnum 126
ndate 06/15/1986
nemp 126
notes
ctype S
ref 3
cnum 20

cdate 06/25/1986
empnum 126
ndate 07/09/1986
nemp 126
notes He likes the D rings
ctype S
ref 3
cnum 21

cdate 12/02/1985
empnum 125
ndate 01/10/1986
nemp 127
notes She'll only need a few now & then
ctype S
ref 4
cnum 7

cdate 01/10/1986
empnum 127
ndate 02/15/1986
nemp 127
notes
ctype S
ref 4
cnum 10

cdate 01/15/1986
empnum 127
ndate 02/20/1986
nemp 127
notes Binders were the wrong size
ctype C
ref 4
cnum 17

cdate 02/20/1986
empnum 127
ndate 03/15/1986
nemp 127
notes She won't need more til march
ctype Q

ref	4
cnum	11
cdate	03/17/1986
empnum	127
ndate	04/15/1986
nemp	127
notes	
ctype	S
ref	4
cnum	12
cdate	04/15/1986
empnum	127
ndate	06/01/1986
nemp	126
notes	I'll be on vacation for her next order
ctype	S
ref	4
cnum	13
cdate	12/06/1985
empnum	126
ndate	01/15/1986
nemp	126
notes	They might be a retail outlet
ctype	S
ref	5
cnum	14
cdate	01/15/1986
empnum	126
ndate	02/12/1986
nemp	126
notes	bring some samples by
ctype	S
ref	5
cnum	15
cdate	02/20/1986
empnum	125
ndate	05/01/1986
nemp	126
notes	
ctype	S
ref	5
cnum	16

The product Table

pcode	price	descrip
10d	\$1.55	1.0" D-ring (special order)
10r	\$1.35	1.0" regular binder
15d	\$1.97	1.5" D-ring binder
20d	\$2.32	2.0" D-ring binder
25d	\$2.65	2.5" D-ring binder
25r	\$2.41	2.5" regular binder
35d	\$4.75	3.5" D-ring binder
40r	\$4.98	4.0" regular binder

The prospect Table

fname	Paul
lname	Machado
title	Mr.
company	GATEWAY CHARTERS, INC.
add1	365 Notre Dame Drive
add2	
add3	
city	San Jose
state	CA
zip	95102
phone	408/555-1212
source	R
ref	1

fname	Mike
lname	Pitts
title	Dr.
company	TESTING ENGINEERS, INC.
add1	1536 Fordham Court
add2	
add3	
city	Palo Alto
state	CA
zip	94306
phone	415/555-1212
source	B
ref	2

fname	John
lname	Pitzer
title	Mr.
company	PITZER SHIRT SALES, INC.

add1 248 Sealle Avenue
add2
add3
city San Bruno
state CA
zip 94721
phone 415-555-1212
source P
ref 3

fname Linda
lname Roos
title Ms.
company THE DESIGN ASSOCIATES
add1 626 El Camino Real
add2
add3
city San Carlos
state CA
zip 94308
phone 415/555-1212
source N
ref 4

fname Jocelyn
lname McCaffrey
title Ms.
company HEALTHWAY NATURAL FOODS
add1 405 Fernando Avenue
add2
add3
city Palo Alto
state CA
zip 94306
phone 415/555-1212
source O
ref 5

The sale Table

cnum	pcode	quantity	discount
1	25d	5	0
2	25r	500	40
3	35d	45	15
5	15d	68	22
6	40r	45	15
7	25r	10	5
8	25d	125	40
9	25d	75	15
10	10r	10	5

Appendix D. The leads DATABASE

12	40r	25	10
13	35d	10	5
14	20d	15	10
15	20d	15	10
16	20d	75	25
18	40r	75	25
19	40r	125	30
20	35d	50	20
21	25d	45	20
22	10d	200	45
23	10d	250	45
24	20d	25	15

The sperson Table

empnum 125
position Sales Manager
lname Greenberg
fname Marlene
add1 2525 Market Street
add2 Suite 250
add3
city San Francisco
state CA
zip 94103
phone 415/555-1212

empnum 126
position Western Sales Manager
lname Bell
fname Jerome
add1 675 Sharon Park Drive
add2
add3
city New York
state NY
zip 10023
phone 212/555-1212

empnum 127
position Eastern Sales Manager
lname Fuller
fname Robert
add1 1047 Rich Avenue
add2 P.O. Box 742
add3
city San Francisco
state CA
zip 94104
phone 415/555-1212

empnum	128
position	Area Rep
lname	Kimball
fname	Molly
add1	705 Starbush Way
add2	
add3	
city	Sunnyvale
state	CA
zip	95212
phone	408/555-1212

empnum	129
position	Area Rep
lname	Richardson
fname	Isabelle
add1	201 San Antonio Circle
add2	Suite 405
add3	
city	Menlo Park
state	CA
zip	94315
phone	415/555-1212

empnum	130
position	Area Rep
lname	Wieseman
fname	Frank
add1	365 Avenue of the Americas
add2	Suite 1705
add3	
city	New York
state	NY
zip	10027
phone	212/555-1212

Appendix E

THE STRUCTURE OF THE leads APPLICATION

The following illustration shows the structure of the **leads** application. It shows the functions and report drivers called by each menu selection as well as the forms and reports they open.

Menu Selection	Function or Report Driver	Form or Report
Contact	decide choose prospect contact	f__decide f__decide f__prospect f__contact f__empnum f__sale f__pdesc
Report		
Mailing-labels	rd__label	r__label
Sales	rd__sales	r__sales
Follow-up	rd__follow	r__follow
Letters	rd__letter	r__letter
Update		
Contact	qcontact	f__qcontact
Prospect	decide choose prospect	f__decide f__decide f__prospect
Administration		
Product	product	f__listprod
Salesperson		
Add	sperson	f__sperson
Update	sperson	f__sperson
Delete	sperson	f__sperson
List	rd__sperson	r__sperson

Index

* 87, 105, 126

[] 28

{ } 73, 89

.4gl filename extension 171, 226

.c filename extension 226

.ec filename extension 226

.frm filename extension 80, 84, 89, 225, 226

.o filename extension 226

.per filename extension 80, 171, 225, 226

4GL 1, 19

A

Administration menu 15, 34, 40, 41, 67

AFTER DELETE clause 159, 165

AFTER FIELD clause 30, 151, 152, 153, 159, 162, 164

AFTER GROUP OF subsection 131, 139, 140

AFTER INSERT clause 159, 164, 167

AFTER ROW clause 159, 161, 164, 166

Aggregate function 114

ALTER TABLE statement 77

Analysis, systems 3, 62

ANSI 99

Array

about 32, 86, 138, 148

defining an 148

of records 36, 87, 158

vs. record 149

ARROW key 154, 158

ARR_CURR function 32, 155, 161

Assembly language 1

Asterisk 87, 105, 126

AT clause 125

Attribute

AUTONEXT 83

COMMENTS 84

definition 47

DOWNSHIFT 84

INCLUDE 83

UPSHIFT 83, 84

WITHOUT NULL INPUT 219

ATTRIBUTES section, form 28, 83

Audience 7

AUTONEXT attribute 29, 83

AVG function 114

B

BEFORE DELETE clause 159

BEFORE FIELD clause 31, 151, 152

BEFORE GROUP OF subsection 139

BEFORE INSERT clause 159, 164

BEFORE ROW clause 159, 160, 161, 166

BETWEEN clause 129

Building a form 81, 84, 225

buildleads installation script 222, 223

BY NAME clause 30, 90

C

c4gl command 221

CALL statement 17, 66

Character string 101

CHAR data type 73

CLEAR SCREEN statement 65, 66

CLEAR WINDOW statement 98

CLIPPED clause 128

CLOSE FORM statement 93

CLOSE WINDOW statement 96

COBOL programming language 1

Codd, E. F. 99

Column

data type 73

defined 24, 47

name 73

COLUMN clause 140

COMMAND clause 64, 65, 70

Comment 31, 73, 89, 100

INDEX

COMMENTS attribute 84, 94
Compiling programs 221
Connect privilege 77
CONSTRUCT statement 144, 146, 147
contact function 60, 91, 92, 148, 153, 176
CONTROL keys 35
Conventions 14, 15, 69, 72, 86, 89, 105
Conversion, data 79
COUNT function 114
C programming language 1
CREATE INDEX statement 74
create program 32, 74, 152, 172
CREATE TABLE statement 33, 72
CREATE UNIQUE INDEX statement 74
Creating an index 73
Current window 98
CURRENT WINDOW statement 98
Cursor 102, 135

D

Data
 access method 4
 conversion 79
 type 33, 73
 validation 4, 79
Database management system 1
Database privilege 77
DATABASE section 28, 83
DATABASE statement 86
Data entry form 57
DATE data type 73
Date formats with **PRINT USING** 138
decide function 59, 60, 90, 179
DECLARE statement 102, 135
Default form 81
Default value 29
DEFER INTERRUPT statement 167
DEFINE section 136
DEFINE statement 87, 100, 105, 126, 158
Defining an array 148
Definition
 attribute 47
 column 24, 47
 field 47
 file 47
 null value 217
 record 47
 relation 47

Definition (continued)

 row 24, 47
 table 24, 47
 tuple 47

DELETE statement 36
DELIMITERS 150
DEL key 167
Development environment 4
DISPLAY ARRAY statement 32, 148, 153, 154
DISPLAY FORM statement 90, 93
Display label 120, 135
DISPLAY statement 95, 125
DOWN ARROW key 154
DOWNSHIFT Attribute 84
Driver, report, defined 124
Drop, defined 76
DROP DATABASE statement 76
DROP TABLE statement 77, 134
Dummy menu 65, 66

E

E. F. Codd 99
Edit-checking 29
empty function 65
END FOREACH clause 103, 126
END FUNCTION clause 66, 89, 92
END INPUT clause 152, 156
END MAIN clause 74, 89
END MENU clause 65
ERROR statement 66
Error trapping 165
Execution, program 221
EXIT MENU clause 70

F

FETCH statement 102
Field, defined 47
Field tag 28, 150
File, defined 47
Filename extension 80, 223, 226
FINISH REPORT statement 126, 132, 136, 140
FLOAT data type 73
FOREACH statement 103, 124, 125, 130, 135, 136, 150, 153, 158
Form
 about 4, 19, 57, 79
 ATTRIBUTES section 28, 83
 AUTONEXT attribute 83

Form (continued)

- COMMENTS attribute 84
- DATABASE section 28, 83
- default 81
- DELIMITERS 150
- DOWNSHIFT attribute 84
- field 79
- INCLUDE attribute 83
- INSTRUCTIONS section 28, 83, 150
- SCREEN section 28, 83
- TABLES section 28, 83
- UPSHIFT attribute 83, 84
- WITHOUT NULL INPUT attribute 219
- form4gl** 81, 84, 225
- Format 72, 100
- FORMAT section 16, 17, 128, 136
- FORMONLY 150
- FOR statement 140
- Fourth generation language 1
- FROM clause 100, 105
- Function
 - about 66, 88
 - CALL statement 66
 - END FUNCTION clause 66
 - FUNCTION statement 66, 88, 89
- Function key 35, 154, 158, 159, 161, 164
- FUNCTION statement 66, 88, 89
- f_contact** form 92, 191
- f_decide1** form 61, 191
- f_decide2** form 192
- f_decide3** form 192
- f_empnum** form 193
- f_listprod** form 193
- f_pdesc** form 194
- f_prefix** 90
- f_prospect** form 62, 82, 83, 194
- f_qcontact** form 23, 195
- f_sale** form 95, 196
- f_scontact** form 20, 210
- f_sperson** form 59, 197
- f_sproduct** form 34, 210

G

- Generations, programming language 1
- globals.4gl** file 86, 89, 152, 182
- GLOBALS statement 86
- Global variable 86
- GRANT statement 77

- GROUP BY clause 114, 115, 120, 135, 139
- GROUP SUM clause 131

H

- Help 4, 12
- Help line 69
- Help message 63

I

- IF statement 133, 156, 163, 165, 167
- INCLUDE attribute 29, 83
- Index

- and joins 73
- creating an 73
- unique 74
- when to create 73

- Information retrieval 73

- INITIALIZE statement 217

- Input 19

- INPUT ARRAY statement

- about 33, 35, 157, 160
- AFTER DELETE clause 159, 165
- AFTER FIELD clause 159, 162, 164
- AFTER INSERT clause 159, 164, 167
- AFTER ROW clause 159, 161, 164, 166
- ARR_CURR function 161
- BEFORE DELETE clause 159
- BEFORE INSERT clause 159, 164
- BEFORE ROW clause 159, 160, 161, 166
- function key 164
- ON KEY clause 161, 162
- SCR_LINE function 161
- WITHOUT DEFAULTS clause 35, 160

- INPUT BY NAME statement 31

- INPUT statement

- about 90, 91, 151, 152, 155
- AFTER FIELD clause 30, 151, 152, 153
- BEFORE FIELD clause 31, 151, 152
- BY NAME clause 30, 90
- END INPUT clause 152
- WITHOUT DEFAULTS clause 90

- INSERT statement 35, 91, 156, 165

- Installing the leads database 222, 223

- INSTRUCTIONS section, form 28, 150

- INTEGER data type 73, 76

- Interrupts, trapping 167

- INTO clause 27, 100, 105, 150, 158

- INTO TEMP clause 120, 134

INDEX

int_flag 168

IS NOT NULL clause 219

IS NULL clause 218, 219

J

Join

about 106, 135

concept 50

condition 50

outer 147

Joining table 106

K

Key 49

KEY clause 65

Keywords 72

L

Label 127

Language generations 1

leads menu system 40, 66

LET statement 91, 133, 145, 150

LIKE clause 86, 87, 103, 126, 149

Link 221

Local variable 89

M

Mailing label report 42

Main routine 66, 88

MAIN statement 74, 88, 89

Maintenance 2, 87, 103

MAX function 114

MDY function 134

Menu 4, 12, 40, 60, 63, 64

menu program 26, 174

MENU statement

about 65, 95

CLEAR SCREEN statement 65

COMMAND clause 64, 65, 70

END MENU clause 65

EXIT MENU clause 70

KEY clause 65

MESSAGE statement 94, 95, 152

MIN function 114

MONEY data type 73

MONTH function 120, 133, 135

N

NOTFOUND 102

NOT NULL clause 74

Null

about 163

in a Boolean expression 163, 217

on input 167

string 126

value 31, 74, 217

value, defined 217

O

On-line help 12

ON EVERY ROW subsection 17, 128, 138

ON KEY clause 160, 161, 162

ON LAST ROW subsection 132, 136, 140

OPEN FORM statement 89, 93

OPEN statement 102

OPEN WINDOW statement

about 93

ATTRIBUTE clause 96

WITH FORM clause 93

Operating system permission 77

ORDER BY clause 27, 118, 129, 131, 135, 150

Outer join 147

OUTPUT section 16, 127, 128, 130, 136

OUTPUT TO REPORT statement 125, 126

P

PAGE HEADER subsection 17, 128, 130,
136, 138

PAGE LENGTH statement 128

PAGENO variable 136

PAGE TRAILER subsection 17, 128, 140

Pascal programming language 1

pa__ prefix 86

Permission 77

PREPARE statement 144, 147

PRINT statement 128, 140

PRINT USING statement 137, 138

Privilege

about 77

connect 77

resource 78

Problem definition 10

product function 87, 159, 160, 183

Programming language generations 1

PROMPT statement 94, 95, 129, 147
prospect function 60, 85, 86, 90, 91, 185
prospect table 87, 91
 Prototype 3, 70
pr__ prefix 86, 105

Q

qcontact function 23, 144, 186
 Query by example 22, 79
 Query by example, defined 144

R

rd_follow report driver 198
rd_label report driver 125, 199
rd_letter report driver 199
rd_sales report driver 133, 200
rd_sperson report driver 202

Record

about 36, 87
 defined 47
 variable 86, 104, 113
 vs. array 149

Relation, defined 47

Relational operator 146

Report

about 42
AFTER GROUP OF subsection 131, 139, 140
BEFORE GROUP OF subsection 139
COLUMN clause 140
 defined 123
DEFINE section 136
 design of 42
 driver 124
FINISH REPORT statement 126, 132, 140
FORMAT section 16, 17, 128, 136
GROUP BY clause 139
GROUP SUM clause 131
ON EVERY ROW subsection 17, 128, 138
ON LAST ROW subsection 132, 140
ORDER BY clause 131
 output, defined 123
OUTPUT section 16, 127, 128, 130, 136
OUTPUT TO REPORT statement 125, 126
PAGE HEADER subsection 17, 128, 130, 136, 138
PAGE LENGTH statement 128
PAGENO variable 136

Report (continued)

PAGE TRAILER subsection 17, 128, 140
PRINT statement 140
rd_label report driver 125
rd_sales report driver 133
REPORT statement 126, 136
REPORT TO clause 128
REPORT TO PRINTER clause 128
r_follow 45
r_label 43, 124, 126
r_letter 45
r_medium 129
r_sales 43, 132
r_sperson 16, 17
 setting up a 124
SKIP TO TOP OF PAGE statement 128
START REPORT statement 125
SUM clause 132
USING clause 132, 137, 140
 writer 1, 4, 15
 writer, defined 123

Report menu 18

REPORT statement 126, 136
REPORT TO clause 128
REPORT TO PRINTER clause 128
 Resource privilege 78
 Retrieval, information 73
RETURN character 72
REVOKE statement 78
 Row, defined 24, 47
r_follow report 45, 203
r_label report 43, 124, 126, 204
r_letter report 45, 205
r_medium report 129
r__ prefix 42
r_sales report 19, 43, 132, 207
r_sperson report 16, 17, 209

S

Salesperson menu 15
 Schema 51, 71
 Screen array 161
 Screen handler 1, 3, 4
 Screen record 32, 154
SCREEN section 28, 83
 Scroll 154
SCR_LINE function 161

SELECT statement
 about 27, 50, 99, 100, 124, 126, 129, 134,
 139, 147, 150, 153
 BETWEEN clause 129
 computation 108
 FETCH statement 102
 FROM clause 100, 105
 GROUP BY clause 135
 INTO clause 27, 100, 105, 158
 INTO TEMP clause 120, 134
 join 106
 OPEN statement 102
 ORDER BY clause 27, 129, 135, 150
 SELECT keyword 27
 WHERE clause 27, 50, 100, 106, 129, 135
SERIAL data type 73, 76, 91, 156
SET clause 36, 92
 Setting up a report 124
SET_COUNT function 153, 158
 Singleton **SELECT** 101
SKIP TO TOP OF PAGE statement 128
SLEEP statement 66
SMALLFLOAT data type 73
SMALLINT data type 73
 Solution design 10, 40
SPACE character 72
sperson function 189
SQL 27, 47, 91, 99
SQLCA.SQLERRD[2] 156
 Square brackets 28
START REPORT statement 125
status variable 102
 Structured language 1
 Structured Query Language 99
 Submenu 67
 Subquery 118, 120
SUM clause 132
SUM function 114, 120, 135
 Systems analysis 3, 62
s_contact function 20
s_contact program 31, 211
s_medium program 213
s_product program 34, 214

T

TAB character 72
 Table, defined 24, 47

TABLES section 28, 83
 Tag, field 28, 150
 Temporary table 28, 118
THRU clause 90
TODAY 29
Top-level menu 13, 66, 69
 Trapping errors 165
 Trapping interrupts 167
 Tuple, defined 47
 Type conventions 72
 Type of data 33

U

Unique identifier 49
 Unique index 74
UP ARROW key 154
Update menu 22
UPDATE statement
 about 36, 92
 SET clause 36, 92
 WHERE clause 36, 92
UPSHIFT attribute 83, 84
 User interface 79
USING clause 121, 132, 137, 138, 140

V

Validation, data 79
 Variable 85
 Variable, local 89

W

WHENEVER ERROR CONTINUE statement 164
WHENEVER ERROR STOP statement 165
WHERE clause 27, 36, 37, 50, 92, 100, 101,
WHILE statement 102
 Wildcard 79, 146
 Window 3, 4, 20, 57, 79, 92
WITHOUT DEFAULTS clause 35, 90, 160
WITHOUT NULL INPUT attribute 219
wwait function 17, 140, 190
w_empnum window 93
w_pdesc window 22
w_sale window 20, 95

Y

YEAR function 120

BUILDING APPLICATIONS USING A 4GL

BUILDING APPLICATIONS USING A 4GL is the first book that explains, with step-by-step examples, how you can take advantage of the power of a fourth generation programming language. It takes you through all the steps of building an actual 4GL application, from the systems analysis phase, through the prototype phase, and finally through the complete, detailed code.

BUILDING APPLICATIONS USING A 4GL draws its examples from **INFORMIX-4GL**, which is based on the industry-standard SQL database language. Chapter 6 is devoted exclusively to SQL's powerful **SELECT** statement. Other standard SQL statements are used throughout the book.

BUILDING APPLICATIONS USING A 4GL covers all the important features of fourth generation languages: the theory of relational database management systems, menu building facilities, screen handling routines including scrolling, report writers, and ad hoc queries using query by example.

BUILDING APPLICATIONS USING A 4GL will give you the in-depth knowledge you need to evaluate and begin using a 4GL effectively.

About the author: Mr. Sobell, author of the best selling **A PRACTICAL GUIDE TO THE UNIX SYSTEM** and **A PRACTICAL GUIDE TO UNIX SYSTEM V** has been programming for more than 20 years. He is the principal of Sobell Associates, an independent consulting firm in the San Francisco Bay Area, specializing in systems analysis, application building, UNIX Shell programming, and technical writing. In addition, Mr. Sobell teaches, lectures, and writes for various publications.